2007

Supporting Views in Data Stream Management System

Thanaa M. Ghanem

Walid G. Aref
Purdue University, aref@cs.purdue.edu

Ahmed K. Elmagarmid
Purdue University, ake@cs.purdue.edu

Per-Ake Larson

Report Number:
07-024

SUPPORTING VIEWS IN DATA STREAM MANAGEMENT SYSTEM

Thanaa M. Ghanem
Walid G. Aref
Ahmed K. Elmagarmid
Per-Ake Larson

CSD TR #07-024
November 2007

Supporting Views in Data Stream Management System

THANAA M. GHANEM, WALID G. AREF, AHMED K. ELMAGARMID
Purdue University
and
PER-ÅKE LARSON
Microsoft Research

In Relational database management systems, views supplement basic query constructs to cope with the demand for "higher-level" views of data. Moreover, in traditional query optimization, answering a query using a set of existing materialized views can yield a more efficient query execution plan. Due to their effectiveness, views are attractive to data stream management systems.

In order to support views over streams, a data stream management system should employ a query language that allows query composition - composition means the ability to compose complex queries from simpler queries. Prior work on languages to express continuous queries over streams has defined a stream as a sequence of tuples that represents an infinite append-only relation. This paper shows that composition of queries, and hence supporting views, is not possible in the append-only stream model. Then, the paper proposes the Synchronized SQL (or SyncSQL) query language that defines a stream as a sequence of modify operations (i.e., insert, update, and delete) against a relation with a specified schema. Inputs and outputs in any SyncSQL query are interpreted in the same way and, hence, SyncSQL expressions can be composed. An important issue in continuous queries over data streams is the frequency by which the answer gets refreshed and the conditions that trigger the refresh. Coarser periodic refresh requirements are typically expressed as sliding-window queries. In this paper, the sliding-window approach is generalized by introducing the synchronization principle that empowers SyncSQL with a formal mechanism to express queries with arbitrary refresh conditions. After introducing the semantics and syntax, we lay the algebraic foundation for SyncSQL and propose a query matching algorithm for deciding containment of SyncSQL expressions.

Efficient execution of continuous queries is a key requirement in streaming applications. Hence, this paper introduces the Nile-SyncSQL prototype server to support SyncSQL queries, and hence supports views over streams. Nile-SyncSQL employs a pipelined incremental evaluation paradigm in which the query pipeline consists of a set of differential operators. We develop a cost model to estimate the cost of SyncSQL query execution pipelines. The cost model is based on estimating the number of tuples that are processed by the various operators in the pipeline. The cost model is used to choose the best execution plan from a set of different plans for the same query (or the same set of queries).

We conduct an experimental study to evaluate the performance of the proposed Nile-SyncSQL prototype server. The experimental results are twofold: (1) showing the effectiveness of the proposed Nile-SyncSQL framework to support continuous queries over data streams; and (2) validating the proposed cost model and showing significant performance gains when views are enabled in data stream management systems.

Categories and Subject Descriptors: ... [**Data Stream Management Systems**]: ...

General Terms: Views, Language, Experimentation, Performance

Additional Key Words and Phrases: Data Streams, Query language, Expression matching, Incremental evaluation

1. INTRODUCTION

Relational database management systems have been used to provide efficient query answering mechanisms over stored data. Views on database tables provide a basic query construct to cope with the demand for "higher-level" views over the base data. Moreover, answering queries using materialized views can yield more efficient query execution plans. In the same time, the emergence of data streaming applications calls for new data management technologies to cope with the characteristics of *continuous* data streams. Examples of data streaming applications include: environmental and road traffic monitoring through sensors [Szewczyk et al. 2004; Yao and Gehrke 2003], online data feeds [Chen et al. 2000], and on-line analysis of network traffic [Cranor et al. 2003; Lerner and Shasha 2003]. Due to their effectiveness, views are attractive to data stream management systems. In this paper, we investigate how to extend data stream management systems by the capability to support views over data streams.

1.1 Views over Streams

Views have been widely used in database management systems. A view defines a function from a set of base tables to a derived table. The derived table (or the view) can be used as input to other functions or queries. Views are needed because the actual schema of the database is usually normalized for various reasons and queries are more intuitive using one or more denormalized relations that better represent the real world. Then, defining a new relation as a view allows queries to be intuitively specified [Gupta and Mumick 1999]. Thus, views supplement basic query constructs to cope with the demand for "higher-level" views of data. Materialized views are proposed as an extension over views. Basically, a materialized view is a view that is materialized by storing its contents in the database. Consequently, query access to the materialized views can be much faster than recomputing the view with every access. Answering a query using a set of existing materialized views can yield a more efficient query execution plan [Halevy 2001].

The emergence of data streaming applications calls for new data management technologies that cope with the characteristics of *continuous* data streams. A data stream is defined as a continuous sequence of tuples. Unlike traditional snap-shot queries over data tables; queries over data streams are continuous. A continuous query is issued once and may remain active for hours or days. The answer to a continuous query is constructed progressively as new input stream tuples arrive. To support views over data streams means the ability to express derived streams as a function of one or more input streams. The derived streams are then used as inputs to other continuous queries. To support views is an attractive property for data stream management for the following reasons:

—**More intuitive query expressions:** Data streams are usually received from a distributed set of data sources (e.g., sensors). A query is more intuitive if expressed using a derived stream (or a view) that represents the real world better. The view can be expressed as a function over one or more input streams. For example, a view may be as simple as the union of two input streams. On the other hand, for more powerful stream management, more complex views should be supported. A complex view may include join or window functions over several

base streams.

—**Answering multiple (concurrent) continuous queries using views:** Views can be beneficial in streaming environments that are characterized by a large number of concurrent overlapping queries. For a set of overlapping queries, a view can be defined to represent the overlapped part among the queries. Each of the overlapped queries can then be reexpressed using the shared view. The shared execution of the overlapped part can be beneficial in optimizing resource consumption during query execution.

—**Data privacy:** An input stream may contain attributes or tuples that should not be seen by a certain group of users. Restricted access to stream attributes can be achieved by defining a view that projects out the private attributes. Then, users are given access only to the view. Multiple views can be defined depending on the privileges of the different user groups.

In order to support views over streams, a data stream management system should employ a *closed* (or composable) continuous query language. A closed query language is a language in which query inputs and outputs are interpreted in the same way, hence allowing query composition. Query composition means the ability to express a query in terms of one or more sub-queries (or views). In this paper, we show that query languages in the streaming literature are not always closed, and hence are not able to support views over streams. Then, we propose the Synchronized SQL query language (SyncSQL for short); a *closed* query language that enables supporting views over streams. We introduce the Nile-SyncSQL prototype server that supports SyncSQL, and hence supports views over data streams. We evaluate the performance of Nile-SyncSQL via an extensive set of experiments. The experimental results illustrate that views over streams have a tremendous effect on the performance of a data stream management system.

1.2 New Challenges to Continuous Query Languages

Query languages in the streaming literature (e.g., [Arasu et al. 2006; Carney et al. 2002; Chandrasekaran et al. 2003; Cranor et al. 2003; StreamSQL ; ESL]) define a stream as a sequence of tuples that represents an infinite append-only relation. Languages based on the append-only model are not *closed*, that is, the result of a query expression is not necessarily an append-only relation. Not being *closed* has the negative effect that query expressions cannot be freely composed, i.e., one cannot express a query in terms of one or more sub-queries. Composition is a fundamental property of query languages (e.g., SQL), and it requires that query inputs and outputs be interpreted in the same way. To support continuous query composition, and hence to support views over streams, the following challenges need to be addressed by continuous query languages.

Challenge 1- Using streams to represent the output of continuous queries that produce non-append-only output: A continuous query may not be able to produce an append-only output relation even when the input streams represent append-only relations. For example, consider an application that monitors a parking lot where two sensors continuously monitor the lot's entrance and exit. The sensors generate two streams of identifiers, say S_1 and S_2, for vehicles entering and exiting the lot, respectively. A reasonable query in this environment is

P_1: *"Continuously keep track of the identifiers of all vehicles inside the parking lot"*. The answer to P_1 is a *view* that at any time point, say T, contains the identifiers of vehicles that are inside the parking lot. S_1 can be modeled as a stream that inserts tuples into an append-only relation, say $\Re(S_1)$, and similarly, S_2 inserts tuples into the append-only relation $\Re(S_2)$. Then, P_1 can be regarded as a *materialized view* that is defined by the set-difference between the two relations $\Re(S_1)$ and $\Re(S_2)$. As tuples arrive into S_1 and S_2, the corresponding relations are modified, and the relation representing the result of P_1 is updated to reflect the changes in the inputs. The result of P_1 is updated by *inserting* identifiers of vehicles entering the lot and *deleting* identifiers of vehicles exiting the lot. Notice that although the input relations in P_1 change by only inserting tuples (i.e., are append only), the output of P_1 changes by both insertions and deletions. The deletions in P_1's output are due to the set-difference operation. P_1's output cannot be represented as an append-only stream. In order to represent P_1's output as a stream, we should be able to represent two different types of stream tuples (one type of stream tuples to represent the *insertions* in the output and the other type of stream tuples to represent the *deletions*).

The commonly used sliding-window model is another source of deletions in the outputs of queries over streams [Arasu et al. 2006]. Tuples need to be deleted from the output of a sliding-window query because input tuples expire as the window slides.

Challenge 2 - Similar interpretation of query inputs and output: To enable query composition, query inputs and output should be interpreted in the same way so that the output of one query can be used as input to another query. Similar interpretation of query inputs and output is not always possible in the append-only stream model. For example, the output of query P_1 can be produced either as (1) a *complete* answer, or as (2) an *incremental* answer. In the case of a complete answer (case 1), at any time point T, the issuer of P_1 sees a state, i.e., a relation containing identifiers of all vehicles inside the lot at time T. In the case of an incremental answer (case 2), the issuer of P_1 receives a stream that represents the changes (i.e., insertions and deletions) in the state. The output in the incremental case is interpreted in the same way as the inputs, namely, as a stream that represents modifications to an underlying relation. However, P_1's incremental answer cannot be *produced* or *consumed* by a query in a language that models a stream as an append-only relation. Existing languages may produce output streams from P_1 but the output streams are interpreted differently from the input streams. For example, the output may be modeled as a stream representing a concatenation of serializations of the complete answer (e.g., RStream in CQL [Arasu et al. 2006], and the output of window queries in TelegraphCQ [Chandrasekaran et al. 2003]). Alternatively, CQL divides the output into two append-only streams such that one stream represents the insertions in the output while the second stream represents the deletions (i.e., IStream and DStream).

Consider the following query, P_2, from the same application: *"Group the vehicles inside the parking lot by type (e.g., trucks, cars, or buses). Continuously keep track of the number of vehicles in each group"*. By analyzing the two queries, P_1 and P_2, it is obvious that P_2 is an aggregate query over P_1's output. This observation

motivates the idea of defining P_1 as a view, say V_1 and then, expressing both P_1 and P_2 in terms of V_1. Answering P_2 using views requires a language that allows query composition.

Challenge 3 - Expressing general refresh conditions (other than time- or tuple-based refresh conditions): Another important issue in data stream query languages is the frequency by which a query answer gets refreshed as well as the conditions that trigger the refresh. In streaming applications with high tuple arrival rates, an issuer of continuous queries may not be interested in refreshing the answer in response to every tuple arrival. Instead, coarser refresh periods may be desired. For example, instead of reporting the count of vehicles with every change in the parking lot, P_2 may be interested in updating the count of vehicles in each group *every four minutes*. This refresh condition is temporal. However, a powerful language should allow a user to express more general refresh conditions based on time, tuple arrival, events, relation state, etc. For example, P_2 may be interested in updating the count of vehicles in each group whenever a police car enters the parking lot. In this case, the refresh condition is event-based where the event is defined as "the entrance of a police car".

Challenge 4 - Expressing queries over streams that do not represent append-only relations: A general purpose continuous query language should be able to express queries over streams that follow models other than the append-only model. Streams from different domains may be interpreted differently by different applications [Maier et al. 2005]. For example, one sequence of tuples (or one stream) can represent an infinite append-only relation (e.g., S_1 in P_1). On the other hand, another sequence of tuples may represent an update stream in which an input tuple is an update over the previous tuple with the same key value. For example, consider a temperature-monitoring application in which sensors are distributed in rooms and each sensor continuously reports the room temperature. A reasonable query in this environment is T_1: *"Continuously keep track of the rooms that have a temperature greater than 80"*. Neither the input nor the output streams in T_1 represent append-only relations. The input in T_1 is an update stream in which a room identifier is considered a key and an input tuple is an update over the previous tuple with the same key value. Notice that although an update stream is also represented as a sequence of tuples, the interpretation of an update stream is different from the interpretation of an append-only stream. The output tuples from T_1 represent an incremental answer that includes insertions and deletions for rooms that switch between satisfying and not satisfying the query predicate.

We can summarize the limitations of the existing continuous query languages as follows.

(1) Cannot always compose queries, and hence cannot support views, because of the different interpretation or the division of the output streams.

(2) Cannot express queries over streams that do not adhere to the append-only relation representation.

(3) Cannot produce incremental answers for queries that do not produce an append-only output.

(4) Refresh conditions are restricted to be either time- or tuple-based.

1.3 Illustrative Example

In this section, we give an example to illustrate that query composition, and hence views, cannot be supported by a language that restricts the stream definition to the append-only model. Consider the following query, P_3, from the same application as that of P_1 in Section 1.2. P_3: "*Continuously keep track of the identifiers of all vehicles inside the parking lot,* ***report changes in the answer every 2 minutes***". In the following, we use CQL [Arasu et al. 2006] as a representative for the class of languages that uses the append-only model. CQL uses sliding windows to express the coarser refresh periods where a sliding window is defined by two parameters, namely range and slide. Assume that the schema of the input streams consists of two attributes, VID that represents the vehicle identifier, and VType that represents the vehicle type (i.e, car, bus, or truck). CQL can express P_3 in four different ways as follows.

—Case 1: **Relational output:**

SELECT R1.VID, $R1.VType$
FROM S_1[range ∞ slide 2] R1 $-$ S_2[range ∞ slide 2] R2

In this case the output of P_3 is a relation (not a stream). The output relation gives the complete query answer and is refreshed every 2 minutes. The output is not incremental, which means that every 2 minutes, the query issuer sees all identifiers of vehicles inside the lot.

—Case 2: **Streamed relational output:**

SELECT RStream(R1.VID, R1.VType)
FROM S_1[range ∞ slide 2] R1 $-$ S_2[range ∞ slide 2] R2

The output in this case is a stream that represents the concatenations of Case-1's output relation. Basically, whenever the output relation is modified (i.e., every 2 minutes), the whole output relation is re-streamed. Notice that the output stream, say S_o, is interpreted differently from the input streams. An input tuple in any of the input streams (i.e., S_1 or S_2) represents an insertion into the corresponding relations (i.e., $\Re(S_2)$ or $\Re(S_2)$). However, a tuple in S_o may represent a repetition for a previous S_o tuple. This tuple repetition takes place because every tuple in Case-1's output relation is re-produced in S_o whenever the query answer changes. For example, vehicles that are inside the lot for more than 2 minutes are reported several times in S_o. Notice that although both Case 1 and Case 2 produce a complete (non-incremental) relation as output, the two cases differ in the way that output relation is reported to the query issuer. Case-1's output relation is stored and the query issuer needs to pull the modified query answer from the stored relation. On the other hand, in Case-2, a new output relation is streamed out (or pushed) to the query issuer whenever the output relation is modified.

—Case 3: **Stream of insertions to the output relation:**

SELECT IStream(R1.VID, R1.VType)
FROM S_1[range ∞ slide 2] R1 $-$ S_2[range ∞ slide 2] R2

The IStream (or insert stream) operation produces a tuple in the output stream whenever a tuple is inserted in the output relation (i.e., whenever a vehicles enters the lot). Notice that because of the slide parameter of length 2, the inserted tuples are accumulated and are produced in the output stream every 2 minutes.

Although IStreams's output stream is incremental, it gives only a *partial* answer for P_3 because it does not include any information about vehicles exiting the lot.

—Case 4: **Stream of deletions from the output relation:**

```
SELECT DStream(R1.VID, R1.VType)
FROM S1[range ∞ slide 2] R1 − S2[range ∞ slide 2] R2
```

The DStream (or delete stream) operation produces a tuple in the output stream whenever a tuple is deleted from the relation (i.e., whenever a vehicles exits the lot). Notice that because of the slide parameter of length 2, the deleted tuples are accumulated and are produced in the output stream every 2 minutes. DStream's output is incremental but *partial* answer for P_3 because it does not include information about vehicles entering the lot.

Notice that outputs in both Case-1 and Case-2 give a non-incremental answer for P_3. On the other hand, for Case-3 and Case-4, the outputs give an incremental but partial answer for P_3. However, CQL cannot produce a single stream that represents the whole incremental answer to P_3 that include both insertions into and deletions from the parking lot state.

Consider another query P_4:"*Group the vehicles inside the parking lot by type (e.g., trucks, cars, or buses). Continuously keep track of the number of vehicles in each group,* ***report the changes in the answer every 4 minutes***". Careful analysis of P_3 and P_4 shows that: (1) P_4 is an aggregate over the output of P_3, and (2) P_4's refresh time points form a subset of P_3's refresh points. As a result, in a powerful language, P_4 should be easily expressed over the output of P_3. However, none of the four CQL's outputs for P_3 (i.e., Cases 1 to 4) can be used as input to express P_4 for the following reasons:

—Case-1's output is a relation (not a stream) and windows (of range and slide) cannot be expressed over relations. As a result, P_4's sliding window (that slides every 4 minutes) cannot be expressed over Case-1's output relation.

—Case-2's output stream is not incremental and does not represent an append-only relation. However, sliding-window semantics are defined for streams that represent append-only relations. As a result, P_4's window cannot be expressed over Case-2's output stream.

—Both Case-3's and Case-4's output streams represent *partial* answers for P_3. As a result, expressing P_4 over Case-3 (or Case-4) output stream does not give the correct answer for P_4.

—The conclusion is that P_3's incremental output stream should include two different types of tuples to distinguish between the insertions and deletions. However, in the append-only stream model, all stream tuples are of the same type. Hence, the append-only stream model cannot represent P_3's incremental output as a stream.

The discussion in this section shows that the append-only stream model does not allow a language to achieve query composition due to the different interpretation or the division of the query output.

1.4 Nile-SyncSQL: Our Approach to Support Views over Data Streams

This paper presents the Nile-SyncSQL prototype server; an engine to support views over data streams. Nile-SyncSQL is based on the Synchronized SQL query

language; a *closed* language to express composable queries over data streams. Nile-SyncSQL extends the Nile data stream management system [Hammad et al. 2004] to support views over data streams inside Nile. Nile [Hammad et al. 2004] is built using the Predator database management system [Seshadri and Paskin 1997] and the Shore storage manager [Carey et al. 1994]. Nile-SyncSQL has the following distinguishing characteristics:

—**Enables query composition and supports views:** Views is achieved through Synchronized SQL (SyncSQL for short); a *closed* stream query language. In contrast to other languages, SyncSQL defines the stream as a sequence of modify operations (i.e., insert, update, and delete) against a relation with a specified schema. Basically, a continuous query in SyncSQL is semantically equivalent to a *materialized view* where the inputs are relations that are modified by streams of modify operations. The answer to the query is another stream of modify operations that represent changes in the result of the view. The output stream of modify operations is equivalent to incremental maintenance of materialized views [Griffin and Libkin 1995]. The unified representation of query inputs and outputs enables the composition of SyncSQL expressions, and as a result, gives the ability to express and exploit views over streams.

—**Supports generalized refresh conditions:** To cope with the coarser refresh requirement of continuous queries, we introduce the *synchronization principle*. The idea is to formally specify synchronization time points at which the query issuer is interested in receiving an updated query answer. Input tuples that arrive between two consecutive synchronization points are accumulated and are reflected in the output at once at the next synchronization point. The synchronization principle makes it possible to (1) express queries with arbitrary refresh conditions, and (2) formally reason about the containment relationship among queries with different refresh periods.

—**Supports a wider class of streaming applications:** SyncSQL semantics enables Nile-SyncSQL to support a wide class of streaming applications. The generality of SyncSQL is due to the following concepts. (1) General stream model (i.e., as a sequence of insert, update, and delete) that enables modeling a wide variety of streams other than the append-only relation representation, and (2) the synchronization principle that empowers SyncSQL by a mechanism to express general refresh conditions.

—**Uses views for shared execution of continuous queries:** Views can be used as a means for shared execution of continuous queries. For example, an existing view can be used to answer a query if the view matches (a part of) the query. Moreover, the same view can be used to answer several concurrent overlapping queries.

—**Produces an incremental query answer:** The generalized stream model enables Nile-SyncSQL to produce incremental query outputs even for queries that do not produce append-only answer. In the incremental query answer, the query issuer does not receive the whole query answer with every refresh. However, only modifications in the answer are produced. The modifications in the answer can include insertion, update, or deletion of tuples.

1.5 Contributions

The contributions of this paper are as follows:

—We motivate the need for views over streams. We illustrate and give examples to show that views cannot be supported by a language that models a stream as a representation for an append-only relation.

—We propose the SyncSQL query language; a *closed* stream query language that enables views over streams. We define concise semantics, syntax, data types, operators, algebra, and transformation rules for SyncSQL.

—We propose a query matching algorithm to deduce containment relationships among SyncSQL expressions. The query matching algorithm is based on the algebraic foundation of SyncSQL and is used to answer queries using views.

—We give an analytical cost model to estimate the cost of a given SyncSQL execution pipeline. The cost model is based on estimating the number of tuples that are processed by the various operators in the pipeline. The cost model can be used to choose the best execution plan from a set of possible execution pipelines for a given query. The proposed cost model is validated via experiments.

—We design and implement the Nile-SyncSQL prototype to support SyncSQL queries. Nile-SyncSQL extends the Nile data stream management system by adding SyncSQL-specific syntax and operators (e.g., an operator that is responsible for synchronization), developing generalized differential operators to process streams of modifications, and adding a module to control view definition and usage.

—We conduct an extensive experimental study to evaluate the performance of Nile-SyncSQL. The experimental results are twofold: (1) show the effectiveness of our proposed framework to support continuous queries over data streams; and (2) show significant performance gains when views are enabled in data stream management systems.

1.6 Outline

The rest of this paper is organized as follows. Section 2 discusses the related work in both the data stream and database management systems. In Section 3, we give a summary of the queries that are to be used in the rest of the paper. Then, in Section 4, we introduce the semantics and syntax of SyncSQL. Section 4 also includes an extensive set of examples to highlight the features of SyncSQL. The synchronization principle is introduced in Section 5. Section 6 presents the algebraic foundations for SyncSQL. Section 7, introduces a query matching algorithm to deduce the containment relationship among SyncSQL expressions. The design and implementation of Nile-SyncSQL are introduced in Section 8. In Section 9, we present a cost model to estimate the cost of SyncSQL execution pipelines. Nile-SyncSQL is experimentally evaluated and the results are presented in Section 10. Finally, Section 11 concludes the paper and discusses future work.

2. RELATED WORK

In this section, we present an overview of the state-of-the-art data management techniques that are related to views over data streams. In order to support views

over streams, it is necessary to address several aspects of stream processing, e.g., stream query languages, continuous query processing, and shared execution of continuous queries. In the following, we survey related work in the various aspects of stream processing and contrast the related work with the techniques adopted by the `Nile-SyncSQL` prototype.

2.1 Continuous Query Semantics and Languages

A continuous query is a persistent query that is issued once and runs continuously over a period of time. The answer of a continuous query is continuously changing as new tuples arrive to the query input. The unique characteristics of data streams and continuous queries impose new requirements on query languages. In this section, we discuss several proposals for continuous query languages in both the data stream and database literature.

2.1.1 *Continuous Query Languages over Data Streams.* Many research efforts have developed semantics and query languages for continuous queries over data streams, e.g., [Arasu et al. 2006; Bonnet et al. 2001; Carney et al. 2002; Chandrasekaran et al. 2003; Cranor et al. 2003; ESL]. The existing continuous query languages define a stream as a representation of an append-only relation. The append-only stream definition limits the set of queries that can produce streams as output. This is because, even if the input streams represent append-only relations, a continuous query may produce non-append-only output. The non-append-only output cannot then be used as an input stream to other queries, hence limiting query composition.

Stream query languages are required to support stream-specific operations in addition to supporting relational operations (e.g., filtering and join). Window constructs are known to be the most common stream-specific operation. Basically, a window is needed to limit the scope of a query to a window of the most recent input tuples. For example, a time-based sliding-window of size w time units limits the focus of the query to the input tuples that arrive within the last w time units. As time advances, the window slides causing old tuples to expire from the window and new tuple to enter the window. Window queries represent a vital class of queries that produces a non-append-only output. For example, as tuples expire from the input stream's sliding window, the corresponding tuples need to be deleted from the query output, hence resulting in a non-append-only output stream. Different languages follow different approaches to handle the non-append-only output. However, in all languages, the non-append-only output is interpreted differently from the input streams.

In the following, we survey the existing stream query languages and highlight the approaches that are followed to represent non-append-only outputs. Query languages in the streaming literature are classified into two classes: SQL-Like and procedural languages.

SQL-Like Query Languages

The query languages in this class are declarative as they use the same SQL declarative syntax, however, languages in this class use stream-specific semantics.

—**Continuous Query Language-CQL** [Arasu et al. 2006]: CQL is the query language that is used by the STREAM data stream management systems. Query

operators in CQL are classified into three classes, stream-to-relation, relation-to-relation, and relation-to-stream. Window operators belong to the stream-to-relation class that is responsible for transforming a stream to a corresponding relation. Queries are then defined over the input streams' corresponding relations and produce relations as output. Output relations can be transformed to streams via one of three operators: RStream, IStream, or Dstream. The non-append-only output is either divided into two streams or is interpreted in a way that is different from the input stream interpretation. For example, the output stream from the RStream operator represents concatenations of time-varying versions of the output relation and hence cannot be used as input stream to another continuous query. Moreover, the output streams from the IStream or DStream operators cannot be used as inputs to other queries since they represent partial query answers. The CQL proposal does not discuss the semantics of non-unary operators (e.g., join) over two streams with different *slide* parameters.

—**ESL: Expressive Stream Language** [ESL]: ESL is used by the ATLaS data stream management system [ESL]. ESL is designed mainly for data mining and time-series queries. In order to avoid the non-append-only output streams, ESL restricts the set of queries that can produce stream outputs. For example, only unary operators (e.g., selection and projection) can be used to produce output streams. Since a window function produces a non append-only output, window queries produce concrete views as output. A concrete view is basically a table that is stored in the system and is continuously modified as the input changes. At any time point, a query issuer, or an ad-hoc query, can access the stored view to get the current complete answer of the window query. However, concrete views are not streams and cannot be used as inputs to continuous queries. Join is defined between streams and concrete views but the modifications in the view affects only the future join outputs and do not affect the already produced join output. ESL focuses on aggregate queries but does not thoroughly address set-based operators and queries.

—**GSQL** [Cranor et al. 2003]: GSQL is used in the Gigascope stream database that is used for network monitoring. GSQL is mostly a restriction of SQL with some stream-specific extensions. For example, GSQL restricts the join and aggregate functions to be on an ordered (or time) attributes to guarantee that the query will not be blocked. GSQL has a restricted expressibility to guarantee that a query cannot produce a non-append-only output. Sliding-window queries are not supported directly but can be simulated using user-defined functions.

—**StreaQuel** [Chandrasekaran et al. 2003]: StreaQuel is used in the TelegraphCQ stream database system. A StreaQuel query is expressed in SQL syntax and is followed by a for-loop construct to express windows over input streams. The output of a StreaQuel query is a sequence of timestamped sets where each set corresponds to the answer of the query at that time (similar to CQL'S RStream operator). The output stream of sets is not interpreted in the same way as the input stream, hence cannot be used as input to another query.

—**StreamSQL** [StreamSQL]: StreamSQL is a query language that is developed by computer science and data management experts from various universities in conjunction with StreamBase Systems of Lexington, MA [str]. StreamSQL extends

SQL by adding new operations to manipulate streams. For example, StreamSQL defines two join operations, namely tuple join and VJoin, to capture the window-per-stream and window-per-operator semantics for the join. The output stream from a StreamSQL query is append-only and does not include delete or update tuples. However, the StreamSQL's language specifications [StreamSQL] does not address how non-append-only query output (e.g., output from an aggregate query or a sliding-window query) is interpreted for query composition purposes. For example, the output from a sliding-window query will not reflect the tuples that expire when the window slides. There is no discussion in [StreamSQL] about how to represent the output of a set-difference query between two streams.

—**COUGAR's query language** [Bonnet et al. 2001]: Cougar is a sensor database management system. Cougar uses an object-based language in which sensors are modeled as abstract data types (ADT) and sensors' data is modeled as time series. Sensor queries are formulated in SQL with some modifications to the language (e.g., including sequence operators). A sensor query defines a view that is persistent during its associated time interval. This persistent view is maintained to reflect the updates on the sensor database. The langauge has an expression, termed *every*(*t*), where t is a parameter to indicate the frequency by which the persistent view needs to be refreshed.

Procedural Query Languages

An alternative to declarative query languages is to let the user specify the data flow. For example, in a procedural language, users construct query plans via a graphical interface by arranging boxes (corresponding to query operators) and joining them with directed arcs to specify data flow, though the system may later re-arrange, add, or remove operators in the optimization phase. There are two procedural query languages in the literature as follows.

—**SQuAl** [Carney et al. 2002]: SQuAl is used by the Aurora stream management systems. A query in SQuAl is expressed via a graphical interface by arranging query operators and joining them with directed arcs to specify data flow. SQuAL uses the window-per-operator semantics in which each join or aggregate operator is assigned a window. SQuAl operators are designed to guarantee that a query produces append-only output.

—**Tribeca** [Sullivan and Heybey 1998]: is an extensible stream-oriented DBMS designed to support network traffic analysis. Similar to Aurora, Tribeca uses a graphical query interface. However, Tribeca presents a new set of stream-specific operators (e.g., qualification, multiplex, demultiplex) that are different from the relational operators.

Summary

Based on the survey of the various continuous query languages, we can identify the following approaches for handling non-append-only output streams:

—**Restricted expressibility:** To guarantee that the output of the query can be incrementally produced as a stream, a language restricts the set of operators that can be used to express queries over data streams. Sliding windows, for example, are not allowed since they produce non-append-only output. Examples

of systems that follow this approach include Cougar [Bonnet et al. 2001], and Gigascope [Cranor et al. 2003].

—**Periodic output streams:** In this approach the output of the query is produced in a *non-incremental* manner by periodically streaming out the output relation. Notice that the non-incremental output stream does not follow the input stream definition and, hence, cannot be used as input to another query. Examples of systems that follow this approach include TelegraphCQ [Chandrasekaran et al. 2003], and the RStream operator in CQL [Arasu et al. 2006].

—**Output relations:** In this approach, the query language does not allow the non-append-only output to be produced as streams. Instead, only relations can be produced from a non-append-only query. This approach is used in ESL [ESL].

—**Divided output:** CQL [Arasu et al. 2006] divides the query into two separate queries such that each query produces an append-only stream. Basically, one query produces a stream, `IStream`, to represent the inserted tuples and the other query produces a stream, `DStream`, to represent the deleted tuples.

The different interpretation or the division of the query output limits the ability of the language to achieve query composition. In this paper, we propose the `SyncSQL` query language that avoids the append-only stream model and allows the output of any continuous query to be produced incrementally in a single stream.

`SyncSQL` belongs to the class of SQL-like languages. An SQL-like language has the following advantages [Stonebraker et al. 2005]: (1) SQL is a widely used standard that is understood by hundreds of thousands of database programmers and is implemented by every database management system, (2) SQL is based on a set of very powerful data processing primitives and is explicit about how these primitives interact, (3) SQL can be easily understood independent from the run-time conditions, and (4) opens the door to benefit from the rich database literature.

2.1.2 *Continuous Queries in Relational Databases.* Continuous queries were used in relational databases before being used over data streams. The following are examples of systems that support continuous queries over database tables:

—**Tapestry** [Terry et al. 1992]: In this system, both inputs and outputs of the continuous query are relations. Although the input relations in Tapestry are append-only, queries may produce non append-only output if the query includes either a reference to the current time (e.g., GetDate()), or a set-difference between two relations. In order to guarantee an append-only output, Tapestry uses a query transformation that transforms a given query into the minimum bounding append-only query. The coarser refresh of the query is achieved via a "*FOREVER DO, SLEEP*" clause where the query is re-executed after every *SLEEP* period.

—**OpenCQ** [Liu et al. 1999]: Both inputs and outputs of the continuous query are relations. However, the input relations can be modified by general modify operations. A continuous query is periodically re-executed and the output is produced as the delta between two consecutive query executions. Triggers are used to schedule the query re-execution.

Our notion of synchronization time points is similar to OpenCQ's triggers and to Tapestry's "FOREVER DO, SLEEP" loop.

2.2 Views in Database Management Systems

Views have been widely used in database management systems. Basically, a view defines a function from a set of base tables to a derived table. Once defined, the view can be used as input to other queries or views. Views are needed because usually the actual schema of the database is normalized for implementation reasons and the queries are more intuitive using one or more denormalized relations that represent the real world [Gupta and Mumick 1999]. A *materialized view* is a view that is materialized by storing the tuples of the view in the database. Materialized views provide fast access to data since the view is computed once and is stored. Then any query can use the stored results without recomputing the view. Materialized views have been widely used in query optimization since answering queries using an existing view yields more efficient query execution plans.

A materialized view becomes out of date when the underlying base relations are modified. Hence, *view maintenance* is the process of updating the view in response to changes in the underlying relations. In most cases, it is wasteful to maintain a view by recomputing it from scratch because only a part of the view changes in response to changes in the base relations [Gupta and Mumick 1999]. Thus, it is usually cheaper to compute only changes in the view to update its materialization. Algorithms that compute changes to a view in response to changes to the base relations are called *incremental view maintenance* algorithms.

View exploitation is the process of making efficient use of materialized views to speed up query processing [Goldstein and Larson 2001]. Basically, a query optimizer examines the possibility of rewriting a given query expression using one or more of the existing materialized views. Given a query expression, an optimizer uses a view matching algorithm to see which one of existing views can be used to rewrite the given expression. The query optimizer then chooses the rewriting that gives the most efficient execution plan.

In `Nile-SyncSQL`, we investigate how to apply the the various materialized concepts (e.g., incremental maintenance and view matching) in data stream management systems. Basically, we extend the materialized view algorithms to work with streams and continuous queries. We extend the query matching algorithm from traditional view exploitation (e.g., [Goldstein and Larson 2001]) by matching the refresh time points in addition to matching the query expression. Moreover, the physical design of `Nile-SyncSQL` execution pipelines follows the incremental maintenance of materialized views [Griffin and Libkin 1995].

2.3 Processing Continuous Queries over Data Streams

The emergence of data streaming applications calls for new query processing techniques to cope with the high rate and the unbounded nature of data streams. A continuous query is a persistent query that is issued once and the query answer is continuously updated to reflect both new tuples entering the answer and old tuples expiring from the answer. New tuples enter the answer due to the arrival of new stream tuples, while an old tuple expires from the answer if the tuple does not qualify the query predicate any more. A sliding-window query is one of the most

popular types of queries over append-only streams [Babcock et al. 2002; Golab and Ozsu 2003]. Sliding-window queries represent a special class of continuous queries in which input tuples expire in a First-In-First-Out order.

2.3.1 *Sliding-window Query Semantics.* A sliding-window query is a continuous query over n input data streams, S_1 to S_n. Each input data stream S_j is assigned a window of size w_j. At any time instance T, the answer of the sliding-window query equals the answer of the snapshot query whose inputs are the elements in the current window for each input stream. At time T, the current window for stream S_i contains the tuples arriving between times $T - w_i$ and T. The same notions of semantics for continuous sliding-window queries are used in other systems (e.g., [Motwani et al. 2003; Terry et al. 1992]). In our discussion, we focus on the time-based sliding window that is the most commonly used sliding window type. Input tuples from the input streams, S_1 to S_n, are timestamped upon arrival to the system (i.e., we use the tuple's transaction timestamp [Snodgrass and Ahn 1985]). The window w_i associated with stream S_i represents the lifetime of a tuple t from S_i. Notice that an input tuple may carry another timestamp that is assigned by the data source (i.e., valid timestamp [Snodgrass and Ahn 1985]). Operating over valid timestamps poses some difficulties because tuples arrive to system out-of-order with respect to the valid timestamps. The unordered arrival of tuples with respect to the valid timestamp is because of one or more of the following reasons [Srivastava and Widom. 2004]: (1) unsynchronized clocks at the various sources, (2) different network latencies from the various sources to the system, and (3) data transmission over a non-order-preserving channel.

Handling timestamps: In a sliding-window query processor, a tuple t carries two timestamps, t's arrival time, TS, and t's expiration time, ETS. Operators in the query pipeline handle the timestamps of the input and output tuples based on the operator's semantics. For example, if a tuple t is generated from the join of the two tuples $t1(TS1, ETS1)$ and $t2(TS2, ETS2)$, then t will have $TS = max(TS1, TS2)$ and $ETS = min(ETS1, ETS2)$.

2.3.2 *Executing Sliding-window Queries over Data Streams.* Two approaches have been conducted to support sliding-window queries in data stream management systems, namely, query *re-evaluation* [Abadi et al. 2003; Abadi et al. 2005] and *incremental evaluation* [Arasu et al. 2006; Ghanem et al. 2007]. In the query *re-evaluation* method, the query is re-evaluated over each window independent from all other windows. Basically, buffers are opened to collect tuples belonging to the various windows. Once a window is completed (i.e., all the tuples in the window are received), the completed window buffer is processed by the query pipeline to produce the complete window answer. An input tuple may contribute to more than one window buffer at the same time. Examples of systems that follow the query re-evaluation method include Aurora [Abadi et al. 2003] and Borealis [Abadi et al. 2005]. On the other hand, in the *incremental evaluation* method, when the window slides, only the changes in the window are processed by the query pipeline to produce the answer of the next window. As the window slides, the changes in the window are represented by two sets of inserted and expired tuples. Incremental operators are used in the pipeline to process both the inserted and expired tuples and

to produce the incremental changes to the query answer as another set of inserted and expired tuples. Examples of systems that follow the incremental evaluation approach include STREAM [Arasu et al. 2006] and `Nile` [Hammad et al. 2004; Ghanem et al. 2007].

2.3.3 *Data Stream Queuing Model.* Data stream management systems use a pipelined queuing model for the incremental evaluation of sliding-window queries [Arasu et al. 2006]. All query operators are connected via first-in-first-out queues. An operator, p, is scheduled once there is at least one input tuple in p's input queue. Upon scheduling, p processes its input and produces output results in p's output queue. The stream SCAN (SSCAN) operator acts as an interface between the streaming source and the query pipeline. In a sliding-window query processor, SSCAN assigns to each input tuple two timestamps: ts, which equals to the tuple arrival time, and Ets, which equals to $ts + w_i$. Incoming tuples are processed in increasing order of their arrival timestamps ts.

Stream query pipelines use incremental query operators. Incremental query operators process changes in the input as a set of inserted and expired tuples and produce the changes in the output as a set of inserted and expired tuples. An algebra for incremental relational operators has been introduced in [Griffin and Libkin 1995] in the context of incremental maintenance of materialized views (expiration corresponds to deletions). In order to process the inserted and expired tuples, some query operators (e.g., Join, Aggregates, and Distinct) are required to store some state information to keep track of all previous input tuples that have not expired yet.

2.3.4 *Incremental Evaluation of Sliding-window Queries.* In this section, we review the *incremental evaluation* approaches for sliding-window queries. Basically, two approaches have been adopted to support incremental evaluation of sliding-window queries, namely, the *input-triggered* approach and the *negative tuples* approach.

The Input-triggered Approach (ITA)

The main idea in ITA is to communicate only positive tuples among the various operators in the query pipeline [Gama and Gaber 2007; Hammad et al. 2003]. Operators in the pipeline (and the final query sink) use the timestamp of the positive tuples to expire tuples from the state. Basically, tuple expiration in ITA is as follows: (1) An operator learns about the expired tuples from the current time T that equals to the newest positive tuple's timestamp. (2) Processing an expired tuple is operator-dependent. For example, the join operator just purges the expired tuples from the join state. On the other hand, most of the operators (e.g., Distinct, Aggregates and Set-difference) process every expired tuple and produce new output tuples. (3) An operator produces in the output only positive tuples resulted from processing the expired tuple (if any). The operator attaches the necessary time information in the produced positive tuples so that upper operators in the pipeline perform the expiration accordingly.

The Negative Tuples Approach (NTA)

The main goal of NTA is to separate tuple expiration from the arrival of new tuples. The main idea is to introduce a new type of tuples, namely negative tuples,

to represent expired tuples [Arasu et al. 2006; Hammad et al. 2004]. A special operator, EXPIRE, is added at the bottom of the query pipeline that emits a negative tuple for every expired tuple. A negative tuple is responsible for undoing the effect of a previously processed positive tuple. For example, in time-based sliding-window queries, a positive tuple t^+ with timestamp T from stream I_j with window of length w_j, will be followed by a negative tuple t^- at time $T + w_j$. The negative tuple's timestamp is set to $T + w_j$. Upon receiving a negative tuple t^-, each operator in the pipeline behaves accordingly to delete the expired tuple from the operator's state and produce outputs to notify upper operators of the expiration.

Invalid Tuples In ITA, expired tuples are not explicitly generated for every expired tuple from the window but some tuples may expire before their *Ets* due to the semantics of some operators (e.g., set-difference). Operators in ITA process invalid tuples in the same way as negative tuples are processed by NTA and produce outputs so that other operators in the pipeline behave accordingly. This means that even in ITA, some negative tuples may flow in the query pipeline.

2.3.5 *Positive and Negative Tuples.* Streams of positive and negative tuples (i.e., insert and delete tuples) are frequently used when addressing continuous query processing [Abadi et al. 2005; Babu et al. 2005; Ganguly et al. 2003; Ghanem et al. 2007]. However, query languages do not consider expressing queries over these modify streams. This conflict between the language and internal streams is the main obstacle in achieving continuous query composition. In this dissertation, we present `Nile-SyncSQL` as the first stream processing engine that unifies the stream definition between the language and the execution model.

2.4 Shared Execution of Continuous Queries

A typical streaming environment has a large number of concurrent overlapping continuous queries. Sharing the query execution is a primary task for query optimizers to address scalability. The current efforts for shared query execution focus on sharing execution at the operator level. For example, shared aggregates are addressed in [Arasu and Widom 2004b] where an aggregate operator is shared among multiple queries with different window *ranges*. The whole set of aggregate functions should be known in advance in order to design the shared aggregate operator in [Arasu and Widom 2004b].

An algorithm to share the execution of window join operators is proposed in [Hammad et al. 2003] where the join execution is shared among queries that are similar in the join predicate but with different window clauses. The join algorithm is implemented via a special join operator that has two additional components: a scheduler and a router. The join's scheduler is responsible for selecting the order by the which the input tuples are joined. On the other hand, the join's router is responsible for delivering the output tuples to the appropriate queries.

NiagraCQ [Chen et al. 2000] proposes a framework to share the execution among SPJ queries. However, the queries addressed by NiagraCQ uses a restricted set of operators and cannot include windows. Moreover, the semantics of the output streams are not discussed for the purpose of query composition. Shared predicate indexing is used in [Chandrasekaran and Franklin 2003] to enhance the performance of a continuous query processor. Again, [Chandrasekaran and Franklin 2003] ad-

dresses the shared execution only for select operators and does not give a general framework for sharing the execution among an arbitrary set of queries.

In this paper, we use views as a means for the shared execution of continuous queries. Sharing the execution through views is distinguished from the existing approaches in that: (1) it does not require the design or the addition of complex window-aware operators. However, views are supported using differential operators which are general and can support the various types of windows. (2) queries are examined for sharing based on a whole query expression not only at the operator level, and (3) it is not restricted to a specific class of queries or operators, however, the same framework can be used to deduce shared execution decisions for any general query with any set of relational operator.

3. SUMMARY OF QUERIES IN THE PAPER

This section introduces the queries that we use in the rest of the paper to demonstrate the semantics and syntax of SyncSQL. The illustrative queries are drawn from three different applications: a parking-lot monitoring application, a room temperature monitoring application, and a road monitoring application. The applications are selected to cover a wide variety of characteristics of streaming applications. Each continuous query is defined by two parts: (1) the query functionality, and (2) the refresh condition that defines the time points at which the output of the query needs to be refreshed.

3.1 Parking-lot Monitoring Application

The first set of queries is drawn from the parking-lot monitoring application that is discussed in Section 1. The goal of this application is to show that the output of a continuous query over streams may not be append-only even if the input streams are append-only.

In the parking-lot monitoring application, there are two sensors that continuously monitor the lot's entrance and exit. The sensors generate two streams of identifiers, say S_1 and S_2, for vehicles entering and exiting the lot, respectively. Both S_1 and S_2 follow the same schema that has two attributes as follows: <VID, VType>, where "VID" gives the vehicle's identifier and "VType" gives the vehicle type (e.g., car, bus, or truck). Table I gives five example queries over the input streams. Queries P_1 and P_3 are similar in the query functionality but differ in the query refresh period. The same is true for P_2 and P_4. P_5 gives an example of an event-based refresh condition.

3.2 Room Temperature Monitoring Application

The room-temperature monitoring application is an application in which input stream tuples represent modifications to the temperatures of the various rooms. The input stream follows a schema of two attributes as follows: <RoomID, Temperature> where "RoomID" gives the room identifier that represents the primary key for the input stream. In other words, an input stream tuple is an update over the previous tuples with the same "RoomID" value. The "Temperature" attribute gives the room's current temperature. The goal of the temperature monitoring application is to show that some data stream applications cannot be supported

Table I. Parking-lot Monitoring Queries.

Query Name	Query Semantics
P_1	Continuously keep track of the identifiers of all vehicles inside the parking lot
P_2	Group the vehicles inside the parking lot by type (e.g., trucks, cars, or buses). Continuously keep track of the number of vehicles in each group
P_3	P_1 while refreshing the output every two time units
P_4	P_2 while refreshing the output every four time units
P_5	P_2 while refreshing the output whenever a police car enters the parking lot

Table II. Temperature Monitoring Queries.

Query Name	Query Semantics
T_1	Continuously keep track of the rooms that have temperature greater than 80
T_2	Continuously keep track of the rooms that have temperature greater than 100
T_3	T_1 while refreshing the answer every two time units
T_4	T_2 while refreshing the answer every four time units
T_5	T_2 while refreshing the answer whenever a room reports a temperature greater than 120

by a query model that assumes the append-only semantics. Table II gives five example queries over the input temperature stream. Notice that T_1 and T_3 are similar in the query functionality but with different refresh requirements. The same is true for T_2 and T_4. Query T_5 gives an example of an event-based refresh condition.

3.3 Road Monitoring Application

The road monitoring application is an application in which sensors are installed at an intersection of two roads to monitor the traffic. The input stream that is reported by the sensors is an append-only stream that reports the vehicles that pass through the intersection. The input stream follows a schema that consists of one attribute, termed <VID>, that reports the identifiers of the passed vehicles. The goal of this applications is to demonstrate sliding-window queries. Table III gives two example queries. The first query R_1 is interested in the identifiers of all vehicles that passed through the intersection in the last five time units. The second query R_2 is an aggregate query that is interested in the number of vehicles that pass through the intersection in the last five time units.

Table III. Road Monitoring Queries.

Query Name	Query Semantics
R_1	Continuously keep track of the vehicle identifiers that are reported in the last five time units
R_2	Continuously report the number of vehicles that are reported in the last five time units

4. STREAM, QUERY, AND VIEW SEMANTICS

4.1 Stream Semantics

A data stream is defined as a sequence of tuples with a specified schema [Arasu et al. 2006; Chandrasekaran et al. 2003; ESL]. The semantics of the stream is application-dependent, that is, the different applications may interpret the same stream in different ways [Maier et al. 2005]. For example, one sequence of tuples may represent an infinite append only relation (e.g., S_1 in the parking lot application as discussed in Section 3.1). On the other hand, another sequence of tuples may represent a concatenation of time-varying states of a fixed size relation (e.g., Case-2's output stream in Section 1). The semantics of a query depends on the semantics of the input streams. Hence, a query language for data streams should first clearly specify the stream semantics, then explain the query operations given the specified stream semantics.

Query languages in the streaming literature model a stream as a representation for an infinite append-only relation [Arasu et al. 2006; Chandrasekaran et al. 2003; ESL]. The append-only stream model has the following limitations: (1) It limits the applicability of the language since the append-only model cannot represent streams from the various domains (e.g., update streams or streams that represent concatenation of the states of a fixed size relation), and (2) the append-only model hinders the ability of the language to achieve query composition since the append-only model cannot represent non-append-only query outputs.

To overcome the limitations of the append-only model, we introduce the *tagged* stream semantics as a model for representing streams in SyncSQL. Basically, SyncSQL distinguishes between two types of streams: *raw* and *tagged*. A *raw* stream is a sequence of tuples that is sent by remote data sources (e.g., sensors). On the other hand, a *tagged* stream is a stream of modify operations (i.e., insert (+), update(u), and delete(-)) against a relation with a specified schema. A raw stream must be transformed into a tagged stream before being used as input in a query. The raw-to-tagged stream transformation is similar to transforming raw data into tables in traditional databases.

The function that transforms a raw stream to a tagged stream is application-dependent. Consider, for example, P_3 in Section 3.1. Since the input streams in P_3 (i.e., S_1 and S_2) represent append-only relations, the tagging function for S_1 (or S_2) is to attach a "+" tag to every input tuple. The output of a SyncSQL query over a tagged stream is another tagged stream. For example, the output of P_3 is a tagged stream with "+" and "-" tuples where a "+" tuple is produced in P_3's output for every vehicle entering the lot and a "-" tuple is produced for every vehicle exiting the lot. P_3's tagged output gives an incremental answer for P_3, and hence, can be

used as input to another query (e.g., P_4). Notice that tagging functions are needed only to transform streams that are sent by remote data sources. However, streams that are internally generated by the query processor (as a result of executing a query) are already tagged.

The tagged stream model enables SyncSQL to be a powerful and a general purpose language for the following reasons: (1) query composition is achieved due to the unified interpretation of query inputs and outputs as tagged streams, and (2) a wider class of applications can be supported since the tagged stream model is general and can represent streams from various domains (e.g., update streams or streams that represent concatenation of the states of a relation).

The tagged stream model can represent update streams as follows. Consider two different temperature-monitoring applications, say $\texttt{Application}_1$ and $\texttt{Application}_2$. Assume each application has a raw input stream with the following schema "<RoomID,Temperature>Timestamp". Assume also that $\texttt{Application}_1$ treats the input stream as an update stream over the temperatures of the various rooms (Application_1 is the application that is discussed in Section 3.2). In this case, RoomID is considered a key and a tuple is considered an update over the previous tuple with the same key value. On the other hand, $\texttt{Application}_2$ treats the input stream as a series of temperature readings and the RoomID attribute is ignored. Given that the two streams have the same schema, the job of the tagging function is to tell the query processor that the two streams are interpreted differently.

In the query processing phase, the transformation (or tagging) function is implemented inside an operator, called Tagger. For example, in $\texttt{Application}_1$, the input stream tuples are correlated based on the key (i.e., RoomID), hence the Tagger needs to keep a list of all the observed key values (i.e., RoomID) so far. In $\texttt{Application}_1$, the output from the Tagger operator is a tagged stream, say RoomTempStr, that consists of *insert* and *update* operations. Notice that in $\texttt{Application}_1$, the functionality of the Tagger operator is similar to that of the MERGE (or UPSERT) operator in the SQL:2003 standard [Eisenberg et al. 2004]. On the other hand, in $\texttt{Application}_2$, the Tagger operator does not need to keep any state since tuples are not correlated. In $\texttt{Application}_2$, the output from the Tagger operator is a tagged stream, say TempStr, that consists of a sequence of *insert* operations.

Defining raw streams: The following is the SyncSQL syntax for defining raw streams:

$$REGISTER\ SOURCE\ <raw-stream-name>\ \ (<schema>)$$
$$FROM\ \ <portnum>$$

where $<raw-stream-name>$ is the name of the stream, $<schema>$ is the schema of the input stream tuples, and $<portnum>$ is the port at which the stream tuples are received. For example, the raw TemperatureSource stream is defined in SyncSQL by the following statement:

$$REGISTER\ SOURCE\ TemperatureSource\ (int\ RoomID,\ int\ Temperature)$$
$$FROM\ \ port5501$$

Defining tagged streams: The following is the SyncSQL syntax for defining tagged streams over raw streams:

$$CREATE\ TAGGED\ STREAM\ <tagged-stream-name>$$
$$OVER\ <raw-stream-name>$$
$$KEY\ <attrname>$$

where $<tagged-stream-name>$ is the name of the tagged stream and $raw-stream-name>$ is the name of the base raw stream. Notice that the raw stream should be defined first before being used in defining a tagged stream. The $<attrname>$ is the name of the attribute (or list of attributes) that represents the primary key of the input stream. Notice that a primary key is required in order to generate a tagged stream of insert (+), update (u), and delete (-) tuples. The primary key is needed in order to correlate the update and delete tuples to their base tuples. For example, the first input tuple with a certain key value k is considered an insert tuple and is tagged with a "+" sign. A subsequent input tuple with the same key value k is considered an update tuple and is tagged with a "u" sign. Notice also that the tagged stream may be interested in only a subset of the raw stream's attributes. In this case, the attributes of interest need to be identified explicitly in the statement that defines the tagged stream. Two tagged streams can be defined over the source TemperatureSource as follows:

```
RoomTempStr: CREATE TAGGED STREAM RoomTempStr
             OVER TemperatureSource
             KEY RoomID

TempStr:     CREATE TAGGED STREAM TempStr
             OVER TemperatureSource
             KEY NULL
```

EXAMPLE 1. This example demonstrates the mapping from the TemperatureSource raw stream to the two tagged streams: RoomTempStr and TempStr. Assume that the following tuples arrive at the TemperatureSource raw stream: "<a,99>1, <b,75>2, <c,80>3, <a,95>4, <b,85>5." The following tuples represent RoomTempStr: "+<a,99>1, +<b,75>2, +<c,80>3, u<a,95>4, u<b,85>5". Notice that <a,99>1 is mapped to +<a,99>1 while <a,95>4 is is mapped to u<a,95>4. The following tuples represent TempStr: "+<a,99>1, +<b,75>2, +<c,80>3, +<a,95>4, +<b,85>5". Notice that all the tuples in TempStr are *insert* tuples.

The complexity of the Tagger operator is application-dependent. The tagging function is very simple in case of streams that represent append-only relations. The tagging function is more complex in the case of update streams because Tagger needs to keep a state in order to correlate the input tuples. However, the size of the Tagger's state has an upper bound that equals to the number of distinct objects. For example, in $\texttt{Application}_1$ the Tagger's state size cannot exceed the maximum number of rooms. Moreover, Tagger does not need to store rooms that do not report temperature updates. In other words, the size of Tagger's state depends on

the update pattern of the underlying stream. Implementing the tagging function as an operator opens the room for the query optimizer to re-order the pipeline and optimize the memory consumption. As we will show in the experimental evaluation in Section 10, the overhead of the tagging transformation can be minimized by merging the functionality of the Tagger operator with the Select operator. For example, in $\texttt{Application}_1$, the Tagger operator can be merged the Select operator so that only qualified rooms are stored in the state. Notice that new applications may require the introduction of new tagging transformations and new tagging syntax. Each new tagging syntax requires the definition and implementation of a new Tagger operator.

The relational view of a tagged stream. The semantics of query operators (e.g., Select and Join) are defined over relations. However, inputs to a SyncSQL query are tagged streams where each stream represents modifications against a relation. Hence, in order to adopt the well-known semantics of relational operators, SyncSQL queries are expressed over the tagged streams' corresponding relations. Notice that streams of insert and delete tuples are frequently used when addressing continuous query processing [Abadi et al. 2005; Babu et al. 2005; Ganguly et al. 2003; Ghanem et al. 2007; Ryvkina et al. 2006]. However, SyncSQL is the first language that addresses query semantics over tagged streams.

Basically, any tagged stream, say S, has a corresponding time-varying relation, termed $\Re$(S), that is continuously modified by S's tuples. An input tuple in a tagged stream is denoted by "Type<Attributes>Timestamp", where Type can be either insert (+), update (u), or delete(-) and Timestamp indicates the time at which the modification takes place. The relational view is modified by the stream tuples as follows: an insert tuple modifies the relation by inserting a new record, an update tuple modifies the relation by changing the attributes of an existing record, while a delete tuple modifies the relation by deleting an existing tuple. $\Re$(S)'s schema consists of two parts as follows: (1) a set of attributes that corresponds to S's Attributes, and (2) a timestamp attribute, termed TS, that corresponds to the Timestamp field of S's tuples. Timestamp is mapped to $\Re$(S) in order to be able to express time-based windows over S as will be discussed in Section 4.4. At any time point, say T, $\Re$(S) is denoted by R[s(T)] and is the relation resulting from applying S's operations with timestamps less than or equal to T in an increasing order of timestamp.

Definition 1. Time-varying relation. A time-varying relation $\Re$(S) is the relational view of a tagged stream S such that $\Re$(S)= R[S(T)] $\forall$ T, where T is any point in time.

Definition 2. The schema of a time-varying relation. If an input tuple in a tagged stream S is denoted by "Type<Attributes>Timestamp", then $\Re$(S)'s schema is as follows: "<Attributes,TS>", where TS corresponds to the Timestamp field of S's tuples.

EXAMPLE 2. This example demonstrates the mapping from RoomTempStr (as defined in Example 1) to a time-varying relation. Figure 1a shows the following input tuples: "+<a,99>1, +<b,75>2, +<c,80>3, u<a,95>4". Figure 1b gives $\Re$(RoomTempStr) with a schema of three attributes: RoomID, Temperature, and TS. Figure 1b shows that, at time 1, $\Re$(RoomTempStr) reflects the insertion of Room

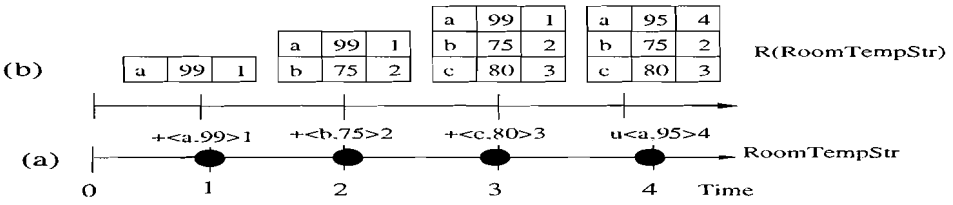


Fig. 1. Illustrating Time-varying Relations.

"a" with temperature 99. At time 4, $\Re$(RoomTempStr) reflects the update of Room "a" temperature to 95.

4.2 Query Semantics

A continuous query over n tagged streams, $S_1 \ldots S_n$, is semantically equivalent to a *materialized view* that is defined by an SQL expression over the time-varying relations, $\Re(S_1) \ldots \Re(S_n)$. Whenever any of the underlying relations is modified by the arrival of a stream tuple, the modify operation is propagated to produce the corresponding set of modify operations in the answer in a way similar to incremental maintenance of materialized views [Griffin and Libkin 1995]. The output of a query can be provided in two forms as follows:
(1) COMPLETE output, where, at any time point, the query issuer has access to a table that represents the complete answer of the query. The answer's table is modified whenever any of the input relations is modified. Notice that the output in this case is non-incremental. (2) STREAMED output, where the query issuer receives a tagged stream that represent the *deltas* (i.e., incremental changes) in the answer.

EXAMPLE 3. This example illustrates the syntax of SyncSQL. We use the keyword STREAMED to indicate that the query asks for an incremental output. The parking lot monitoring query, P_1 from Section 3.1 is expressed as follows:

```
P1 :  SELECT STREAMED R1.VID R1.VType
      FROM ℜ(S1) R1 − ℜ(S2) R2
```

P_1's output is a tagged stream that includes a "+" tuple whenever a vehicle enters the parking lot and a "-" tuple whenever a vehicle exits the lot. P_1 gives an example for expressing queries over append-only streams. As another example for expressing queries over update streams, the temperature-monitoring query T_1, given in Section 3.2, is expressed as follows:

```
T1 :  SELECT STREAMED RoomID, Temperature
      FROM ℜ(RoomTempStr) R
      WHERE R.Temperature > 80
```

EXAMPLE 4. This example demonstrates the execution of a SyncSQL query expression. Assume that the following RoomTempStr's tuples have arrived to T_1 (T_1 is given in Section 3.2): "+<a,78>1, u<a,105>2, +<b,70>3, +<c,95>4, u<a,76>5". Figure 2 gives the input and output streams in T_1. The input tuple +<a,78>1 does not result in producing any output, while the input tuple u<a,105>2 results in *inserting* Room "a" into the answer via the output tuple +<a,105>2 and the input tuple u<a,76>5 results in *deleting* Room "a" via the

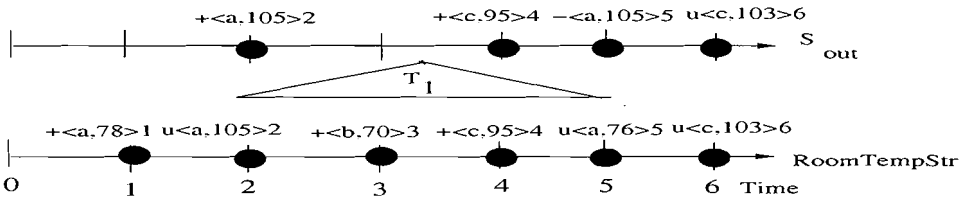


Fig. 2. Example on T_1 Execution.

output tuple -<a,105>5. Other tuples are processed similarly.

4.3 Views over Streams

The unified interpretation of SyncSQL query inputs and outputs enables SyncSQL to exploit *views* over streams. Basically, a view over streams is a function that maps a set of input base streams into an output derived stream. Then, a query can reference the derived stream in a way similar to referencing base streams. Notice that the view is defined once and then can be referenced by any other query if the view's expression is contained in the query's expression. In Section 6, we give an algorithm to deduce the containment relationships among SyncSQL expressions.

EXAMPLE 5. This example demonstrates answering queries using views. As discussed in Section 3.1, P_2 is an aggregate over P_1's output. Hence, we can define a view, say ParkLot, as follows:

```
CREATE STREAMED VIEW ParkLot AS
SELECT R1.VID, R1.VType
FROM ℜ(S1) R1 − ℜ(S2) R2
```

Then, both P_1 and P_2 can be re-written in terms of ParkLot as follows:

```
P1:  SELECT STREAMED P.VID, P.VType
     FROM ℜ(ParkLot) P
P2:  SELECT STREAMED P.VType, Count(P.VID)
     FROM ℜ(ParkLot) P
     GROUP BY P.VType
```

EXAMPLE 6. This example is another demonstration for answering queries using views. Consider the query T_2 from the temperature-monitoring application as explained in Section 3.2. Notice that T_2's selection predicate is contained in T_1's selection predicate. Hence, we can define a view, say $HotRooms_1$, as follows:

```
CREATE STREAMED VIEW HotRooms1 AS
SELECT RoomID, Temperature
FROM ℜ(RoomTempStr) R
WHERE R.Temperature > 80
```

Then, both T_1 and T_2 can be re-written in terms of $HotRooms_1$ as follows:

```
T1:  SELECT STREAMED RoomID, Temperature
     FROM ℜ(HotRooms1) R
T2:  SELECT STREAMED RoomID, Temperature
     FROM ℜ(HotRooms1) R
     WHERE R.Temperature > 100
```

4.4 Window Queries

In this section, we demonstrate the ability of SyncSQL to express sliding-window queries over append-only streams. A sliding window is defined by two parameters as follows: (1) *range* that specifies window size, and (2) *slide* that specifies the step by which the window moves. In existing query languages, windows are defined using special constructs and may be assigned to streams (e.g., [Arasu et al. 2006; Chandrasekaran et al. 2003]) or to operators (e.g., [Carney et al. 2002; ESL]). One limitation of the specific window semantics is that a language that assumes the window-per-stream semantics, for example, cannot express a query with a window-per-operator semantics and vice versa. For example, the window join operator with window of size w is defined in [Carney et al. 2002] as an operator that joins the stream tuples that are within a w units from each other. Such window join operation cannot be expressed by a language that assumes the window-per-stream semantics. In the window join in CQL [Arasu et al. 2006], for example, two windows are defined independently for each input stream.

Unlike other languages, SyncSQL does not assume specific window assignment. Instead, SyncSQL employs the predicate-window model [Ghanem et al. 2006] in which the window *range* is expressed as a regular predicate in the *where* clause of the query. The window's *slide* is expressed using the synchronization principle as will be explained in Section 5. The predicate-window model is a generalization of the existing window models, since all types of windows (e.g., window-per-stream and window-per-operator) can be expressed as predicate windows. For example, a window join (i.e., a window-per-operator) between two streams, S_i and S_j, where two tuples are joined only if they are at most 5 time units apart, can be expressed by the following predicate: $\Re(S_i)$.TS - 5 < $\Re(S_j)$.TS < $\Re(S_i)$.TS+5. Similarly, a time-based sliding window over an append-only stream, say S, (i.e., a window-per-stream) is expressed as a predicate over $\Re$(S)'s TS attribute as shown in the following example.

EXAMPLE 7. Consider the road-monitoring application as described in Section 3.3. Consider also the query R_1 that reports the vehicle identifiers for vehicles that passed through the intersection in the last 5 time units. R_1 is a sliding window query that is essentially a *view* that, at any time point T, contains the identifiers of vehicles that are reported between times T - 5 and T. Such window view is expressed in SyncSQL as follows:

```
CREATE STREAMED VIEW FiveUnitsWindow AS
SELECT *
FROM ℜ(S) R
WHERE Now − 5 < R.TS ≤ Now
```

The view FiveUnitsWindow is refreshed when either $\Re$(S) is modified or Now is changed. Notice that although the input stream S is append-only, *delete* operations are produced in the output to represent expired tuples that fall behind the window boundaries. In Section 5.4 we show that the value of Now can also be represented as a view.

EXAMPLE 8. This example demonstrates query composition by using of FiveUnitsWindow as input to the other road monitoring query R_2. As explained

in Section 3.3, R_2 is an aggregate query that is interested in finding the number of cars that passed through the intersection in the last five time units. R_2 is expressed over FiveUnitsWindow as follows:

```
SELECT STREAMED COUNT(*)
FROM ℜ(FiveUnitsWindow)
```

The output of R_2 is a stream of *update* operations that represent the *incremental* query answer. An *update* operation is produced whenever a vehicle passes the intersection or whenever a vehicle expires from the query answer due to being reported more than 5 time units ago.

5. THE SYNCHRONIZATION PRINCIPLE

If we follow the traditional materialized view semantics, a SyncSQL query answer is refreshed whenever any of the input relations is modified. Unlike materialized views, in streaming applications, modifications may arrive at high rates. A continuous query issuer may be interested in having coarser refresh periods for the answer. For example, as we discuss in Section 3.1, P_3's issuer is interested in getting an update of the answer *every two minutes* independent of the rate of changes in the parking lot state. The coarser refresh periods are achieved using sliding windows in other query languages and are restricted to be either time- or tuple-based [Arasu and Widom 2004a; Chandrasekaran et al. 2003; Li et al. 2005].

In this section, we introduce the synchronization principle as a generalization of sliding windows. The idea of the synchronization principle is to formally specify synchronization time points at which the input stream tuples are processed by the query pipeline. Input tuples that arrive between two consecutive synchronization points are not propagated immediately to produce query outputs. Instead, the tuples are accumulated and are propagated simultaneously at the following synchronization point. The synchronization principle distinguishes SyncSQL by being able to: (1) express queries with arbitrary refresh conditions, and (2) formally reason about the containment relationships among continuous queries with different refresh periods.

5.1 Synchronized Relations

For each input stream in the query, the query issuer specifies time points at which the input stream tuples need to be reflected in the output. Basically, instead of mapping an input stream, say S, into a time-varying relation, S is mapped to a *synchronized relation*, say $\Re_{Sync}$(S). S's tuples are reflected in $\Re_{Sync}$(S) *only* at the time points that are specified by the synchronization stream, Sync. Notice that $\Re_{Sync}$(S) is of coarser granularity than $\Re$(S).

EXAMPLE 9. This examples illustrates expressing queries with coarser refresh periods. Consider the query T_3 from Section 3.2 that is interested in refreshing the query answer every two time units. To achieve the coarser refresh requirement of T_3, we use the synchronized relation $\Re_{Sync_2}(RoomTempStr)$ as input. The synchronization stream $Sync_2$ is defined as: 0, 2, 4, 6, ... Figure 3 illustrates that $\Re_{Sync_2}$(RoomTempStr) is modified by RoomTempStr tuples every two minutes. For example, at time 1, $\Re_{Sync_2}$(RoomTempStr) is empty and "+<a,99>1" is not inserted in $\Re_{Sync_2}$(RoomTempStr) until time 2. T_3 is expressed as a view, say

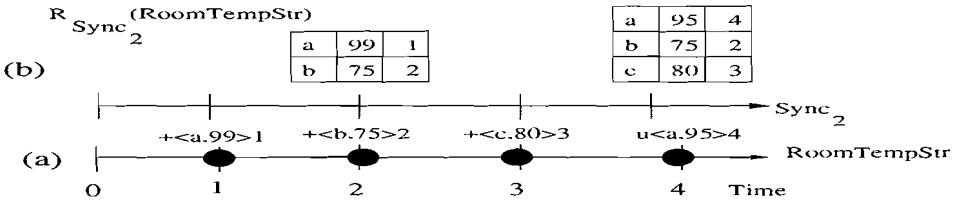


Fig. 3. Illustrating Synchronized Relations.

$\texttt{HotRooms}_2$, as follows:

```
CREATE STREAMED VIEW HotRooms2 AS
SELECT RoomID, Temperature
FROM ℜSync2(RoomTempStr) R
WHERE R.Temperature > 80
```

Notice that $\texttt{HotRooms}_2$ is not refreshed between the synchronization time points. For example, in Figure 3, at time 3, the contents of the relation $\Re_{Sync_2}(\texttt{RoomTempStr})$ is the same as the contents of the relation at time 2 and "+<c,80>3" is not inserted in $\Re_{Sync_2}(\texttt{RoomTempStr})$ until time 4.

EXAMPLE 10. This example demonstrates query composition when using the synchronization principle. Consider the queries P_3 and P_4 from Section 3.1. Since the issuer of P_3 is interested in getting a refresh for the query answer every 2 minutes, we use the synchronization stream $\texttt{Sync}_2$: 0, 2, 4, 6,... to express the view $\texttt{ParkLot}_2$ as follows:

```
CREATE STREAMED VIEW ParkLot2 AS
SELECT R1.VID, R1.VType
FROM ℜSync2(S1) R1 − ℜSync2(S2) R2
```

Then, we use the synchronization stream $\texttt{Sync}_4$: 0, 4, 8,... to express P_4 over $\texttt{parkLot}_2$ as follows:

```
P4:  SELECT STREAMED P.VType, Count(P.VID)
     FROM ℜSync4(ParkLot2) P
     GROUP BY P.VType
```

EXAMPLE 11. This example is another demonstration for query composition when using the synchronization principle. Consider the query T_4 from Section 3.2 that is interested in refreshing the query answer *every 4 minutes*. The coarser granularity T_4 is contained in the view $\texttt{HotRooms}_2$ that is defined in Example 9. The *containment* relationship is decided based on two factors: (1) The selection predicate of T_4 (i.e., *temperature greater than 100*) is contained in $\texttt{HotRooms}_2$'s selection predicate, and (2) T_4's refresh time points (i.e, every 4 minutes) form a subset of $\texttt{HotRooms}_2$'s refresh points. As a result, T_4 can be expressed in terms of $\texttt{HotRooms}_2$.

Example on execution: Figure 4 illustrates the execution of $\texttt{HotRooms}_2$ and T_4 when using the synchronization principle. $\texttt{HotRooms}_2$'s answer is refreshed every two time units. Assume that the following input stream RoomTempStr has arrived at $\texttt{HotRooms}_2$: +<a,105>1, +<b,110>3, +<c,97>4, +<d,75>5, u<a,75>7. In

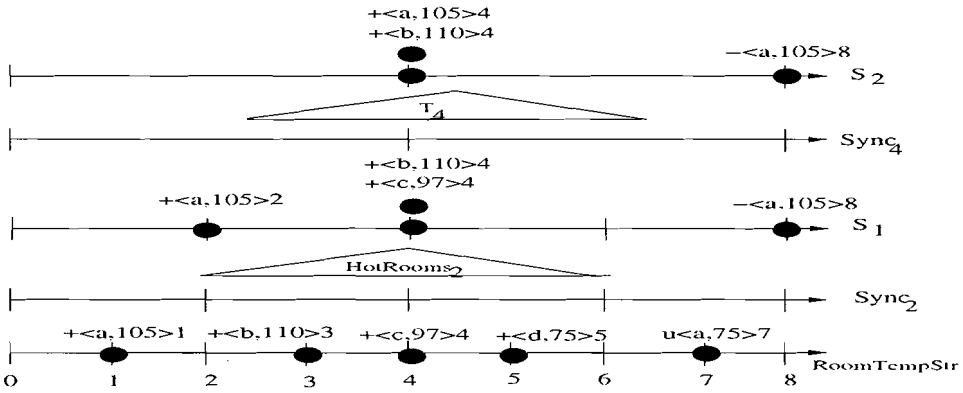


Fig. 4. Example on Answering Queries using Views.

Figure 4, Sync_2 represents HotRooms_2's synchronization stream while S_1 represents HotRooms_2's output. The input tuple +<a,105>1 that arrives at time 1 results in producing the tuple +<a,105>2 at time 2, which is the first synchronization time point after 1. Similarly, +<b,110>3 results in producing +<b,110>4, and u<a,75>7 results in producing −<a,105>8.

Query composition: S_1 is used as input to T_4, which uses the synchronization stream Sync_4: 0, 4, 8, ... S_2 represents T_4's output. As a result, tuple +<a,105>2 that arrives to T_4 at time 2 results in producing the tuple +<a,105>4 at time 4 in S_2. Other tuples are processed similarly by T_4's pipeline.

5.2 Discussion

The idea of accumulating the tuples of an input stream and propagating them in the query pipeline at once is similar in spirit to the idea of heartbeats [Srivastava and Widom. 2004]. In [Srivastava and Widom. 2004], a heartbeat is defined as a special type of tuples that are embedded in the stream such that, at any instant, a heartbeat τ for a set of streams provides a guarantee to the system that all tuples arriving on those streams after that instant will have a timestamp greater than τ. If the stream sources do not provide heartbeats, the data stream management system needs to deduce them based on the given stream characteristics.

The context and objectives of heartbeats is totally different than those of the synchronization time points. Basically, heartbeats are low-level constructs that are automatically generated by the query processor based on the underlying stream characteristics [Srivastava and Widom. 2004]. In other words, the query issuer has no control over the generation of the heartbeats. The goal of heartbeats is to re-order the input stream tuples and to produce an ordered query output. Basically, changing the heartbeats does not change the semantics nor the output of a given query. On the other hand, synchronization is a high-level concept that is expressed by the query issuer through the query language. Unlike heartbeats, the synchronization principle affects the semantics of a query since the same query has different outputs under different synchronization time points. In other words, the synchronization principle and heartbeats are orthogonal, which means that the query processor can use heartbeats in order to generate a correct output for a given

SyncSQL query.

5.3 Synchronization Streams

Before proceeding to the algebraic foundation of SyncSQL, this section discusses synchronization streams in more detail. A synchronization stream (e.g., $Sync_2$) specifies a sequence of time points. However, a synchronization stream is represented and is treated as a tagged stream. The tagged representation of a synchronization stream is characterized by the following. (a) The underlying stream schema has only one attribute, termed TimePoint, and (b) tuples in the stream are *insert* operations where a tuple of the form "+<TimePoint>Timestamp" indicates a synchronization time of value TimePoint where TimePoint = Timestamp. Like any other stream, a synchronization stream Sync has a corresponding time-varying relation $\Re$(Sync). The default clock stream, clockStr: +<0>0, +<1>1, +<2>2, +<3>3, ..., is the finest granularity synchronization stream. Coarser synchronization streams can be constructed using SyncSQL expressions over clockStr.

EXAMPLE 12. The synchronization stream that has a tick every i time points (e.g., i=2 for $Sync_2$) is constructed from clockStr as follows:

```
CREATE STREAMED VIEW Sync_i AS
SELECT C.TimePoint
FROM ℜ(clockStr) C
WHERE C.TimePoint mod i = 0
```

For i=2, a tuple is produced in the output of $Sync_2$ whenever an input tuple, say c, is inserted in $\Re$(clockStr) and c.TimePoint qualifies the predicate "c.TimePoint mod 2 = 0". The output of $Sync_2$ is as follows: +<0>0, +<2>2, +<4>4,..., which indicates the time points: 0,2,4,....

Composition of synchronization streams. The fact that synchronization streams are treated as tagged streams allows SyncSQL to compose synchronization streams to define a larger class of synchronization streams. For example, a synchronization stream can be defined as the *union* or *intersection* of two or more streams.

EXAMPLE 13. The following view expression produces a synchronization stream that is the union of two input synchronization streams (Note that *duplicate elimination* is required so that every time point exists only once in the output stream):

```
create STREAMED view UnionSyncStr as
select DISTINCT(TimePoint)
from ℜ(Sync_2) S_2 UNION ℜ(Sync_5) S_5
```

The output from UnionSyncStr includes a time point T whenever T belongs to either $Sync_2$ or $Sync_5$.

Event-based synchronization. The synchronization principle enables SyncSQL to express queries with event-based refresh conditions. Synchronization streams for event-based conditions can be constructed using SyncSQL expressions as in the following example.

EXAMPLE 14. Consider the query P_5 from Section 3.1 that needs to be refreshed only when a police car enters the parking lot. We use the tagged stream S_1 to generate a synchronization stream, say PoliceSync, such that PoliceSync includes time

points that correspond to the entrance of a police car into the lot. PoliceSync is constructed as follows:

```
CREATE STREAMED VIEW PoliceSync AS
SELECT R.TS
FROM ℜ(S1) R
WHERE R.VType = POLICE
```

An S_1 tuple, of the form "+<VID,VType>Timestamp", results in producing a tuple of the form "+<Timestamp>Timestamp" in PoliceSync's output if "VType" is POLICE. As discussed in Section 4.1, the attribute R.TS reflects the Timestamp attribute of the input stream tuple which corresponds to the time at which a police car is reported in S_1. Notice that, assuming no delays, a police car is reported in PoliceSync's output at the same time instant at which the car is reported in S_1 (i.e., at time Timestamp).

EXAMPLE 15. Consider the temperature-monitoring query T_5 from Section 3.2 that needs to be refreshed only whenever a room reports a temperature greater than 120. We use the tagged stream TempStr, as defined in Example 1, to generate a synchronization stream, say HotSync, such that HotSync includes time points that correspond to reporting a temperature greater than 120. HotSync is constructed using ℜ(TempStr) as follows:

```
create STREAMED view HotSync as
select R.TS
from ℜ(TempStr) R
where R.Temperature > 120
```

An input tuple from TempStr, of the form "+<RoomID,Temperature>Timestamp", results in an output tuple, "+<Timestamp>Timestamp", if "Temperature" is greater than 120. Then, HotSync can be used as a synchronization stream for T_5.

5.4 The Now View

In Example 7, FiveUnitsWindow's contents depend on Now. In order to be consistent with the SyncSQL semantics, Now is defined as a view that is continuously modified by the clock stream clockStr. Notice that ℜ(clockStr) is an append-only relation in which the last inserted tuple indicates the value of Now.

EXAMPLE 16. The following view, NowView, over ℜ(clockStr) always contains the value of Now:

```
CREATE STREAMED VIEW NowView AS
SELECT 1 as KeyAttr, MAX(T.TimePoint) as currTime
FROM ℜ(clockStr) T
```

As discussed in Section 0??, clockStr is a synchronization stream with a schema that consists of one attribute, (i.e., the TimePoint attribute). The NowView is a relation with two attributes, namely KeyAttr and currTime where KeyAttr attribute is set as the primary key of the NowView relation. As a result of the primary key constraint, NowView contains one tuple with key value 1, and the tuple is continuously updated in response to new insertions into ℜ(*clockStr*). The output stream from NowView is as follows: +<1,0>0, u<1,1>1, u<1,2>2, u<1,3>3, ..., where the tuple u<1,i>i, means update the record with KeyAttr value 1, to

have a currTime value i. The view FiveUnitsWindow over stream S from Example 7 is rewritten in terms of NowView as follows:

```
create STREAMED view FiveUnitsWindow as
select R.*
from ℜ(S) R, ℜ(NowView) N
where N.currTime − 5 < R.TS ≤ N.currTime
```

Similar to using clockStr in expressing NowView, the synchronization stream $Sync_2$ can be used to define a view, say SlideTwo. The SlideTwo view contains the value of Now but is updated only every two time units. SlideTwo is used to express a sliding window with slide parameter of size 2.

EXAMPLE 17. This example shows how to use SyncSQL to define a sliding window that is defined by both the *range* and *slide* parameters. Assume we extend the definition of the sliding window in Example 7 such that the window is refreshed every 2 time units instead of every point in time (this corresponds to a sliding window with range 5 units and slide 2 units). In a way similar to using clockStr to define NowView, we use the synchronization stream $Sync_2$ to define a view, say TwoUnitsSlide, as follows:

```
create STREAMED view TwoUnitsSlide as
select 1 as KeyAttr, MAX(T.TimePoint) as currTime
from ℜ(Sync2) T
```

Notice that a new tuple is inserted in $\Re(Sync_2)$ every two time units and the TwoUnitsSlide view consists of only one tuple that represents the latest $Sync_2$ point. Then, the TwoUnitsSlide view can be used to express a sliding window of range 5 and slide 2 over a stream S as follows:

```
create STREAMED view RangeFiveSlideTwo as
select R.*
from ℜSync2(S) R, ℜ(TwoUnitsSlide) N
where N.currTime − 5 < R.TS ≤ N.currTime
```

Only when ℜ(TwoUnitsSlide) is updated to reflect the latest $Sync_2$'s time point, RangeFiveSlideTwo's output is refreshed to include S's tuples that arrived in the 5 time units prior to the latest time point N.currTime. Notice that we used the relation TwoUnitsSlide in expressing the sliding window because TwoUnitsSlide includes ONLY the latest $Sync_2$'s time point. However, the relation $\Re(Sync_2)$ cannot be used in expressing the sliding window because $\Re(Sync_2)$ is append-only and includes all the time points that belongs to $Sync_2$.

6. ANSWERING CONTINUOUS QUERIES USING VIEWS OVER STREAMS

In this section, we lay the algebraic foundation for SyncSQL as the basis for efficient execution of SyncSQL queries. One of our goals while developing SyncSQL is to minimize the extensions over the regular relational algebra. By levering relational algebra, SyncSQL's execution and optimization can benefit from the rich pool of existing techniques for query processing and optimization of traditional databases. Basically, we map continuous queries to traditional materialized views. However, we differentiate continuous queries from materialized views by the synchronization principle.

6.1 Data Types

As discussed in Section 4, although the inputs in SyncSQL expressions are tagged streams, SyncSQL queries are expressed over the input streams' corresponding relations. The output from a SyncSQL expression is another relation that can be mapped into a tagged stream. Basically, a synchronized relation is the main data type over which SyncSQL expressions are expressed. A synchronized relation $\Re_{Sync}(\mathtt{S})$ possesses two logical properties:

—**Data** that is represented by the tuples in the relation, where data is extracted from the input stream S.

—**Time** that is represented by the time points at which the relation is modified by the underlying stream S, where time is extracted from the synchronization stream Sync.

A tuple of the form "+<TimePoint>Timepoint" in the synchronization stream indicates a synchronization time with value TimePoint. Time points along the relation lifetime can be classified into two classes in the following way:

—**Full synchronization points:** A point in time T is termed a full synchronization time point iff $\Re_{Sync_i}(\mathtt{S}_i)$ reflects *all* $\mathtt{S}_i$'s tuples up to time T (i.e., $\Re_{Sync_i}(\mathtt{S}_i)$ is up-to-date with $\mathtt{S}_i$). Basically, the time points $\mathtt{T} \in \mathtt{Sync}_i$ represent the full synchronization points for $\Re_{Sync_i}(\mathtt{S}_i)$.

—**Partial synchronization points:** A point in time T is termed a partial synchronization point if $\Re_{Sync_i}(\mathtt{S}_i)$ does not reflect all $\mathtt{S}_i$ tuples up to time T (i.e., $\Re_{Sync_i}(\mathtt{S}_i)$ is not up-to-date with $\mathtt{S}_i$). Basically, the time points that lies between two consecutive $\mathtt{Sync}_i$ represent the partial synchronization points for $\Re_{Sync_i}(\mathtt{S}_i)$.

The distinction between "*full*" and "*partial*" synchronization points is essential to judge the relationship between the synchronized relation $\Re_{Sync_i}(\mathtt{S}_i)$ and the underlying stream $\mathtt{S}_i$. For example, in Figure 3, time point 2 is a full synchronization point for $\Re_{Sync_2}(\mathtt{RoomTempStr})$ because, at time 2, $\Re_{Sync_2}(\mathtt{RoomTempStr})$ reflects all RoomTempStr's tuples up to time 2. On the other hand, time point 3 is a partial synchronization point for $\Re_{Sync_2}(\mathtt{RoomTempStr})$ because, at time 3, $\Re_{Sync_2}(\mathtt{RoomTempStr})$ reflects RoomTempStr's tuples only up to time 2 but does not reflect the input tuple "+<c,80>3" that arrives at time 3. Thus, at a full synchronization point, the relation is up-to-date with the stream. However, at a partial synchronization point, the relation is not completely up-to-date with the stream. Specifying the relationship between a synchronized relation and the underlying stream is essential for deducing containment relationships among synchronized relations as will be discussed in Section 7.

6.2 Operators

In this section, we discuss the logical SyncSQL operators. The physical operators will be discussed in Section 8. Logical operators in SyncSQL are classified into three classes: Stream-to-Relation (S2R), Relation-to-Relation (R2R), and Relation-to-Stream (R2S). This operator classification is similar to the classification used by CQL [Arasu et al. 2006], but with different instantiations of the operators in each class. The S2R class includes one operator that is used to express the desired

synchronization points. The R2R class includes the traditional relational operators. Finally, the R2S class includes one operator that is used by a query to express the desire of a STREAMED (or an incremental) output.

6.2.1 *The Stream-to-Relation Operator* $\Re$. The same tagged stream can be mapped to different synchronized relations using different synchronization streams. The operator $\Re$ takes a synchronization stream Sync as a parameter and maps an input stream S to a synchronized relation $\Re_{Sync}$(S). As discussed in Section 0??, If an input S tuple is denoted by "Type<Attributes>Timestamp", then $\Re_{Sync}$(S)'s schema is as follows: "<Attributes,TS>", where TS corresponds to the Timestamp field of S's tuples. $\Re$ performs the following: (1) buffers S's tuples, (2) modifies the output relation by the buffered tuples at every Sync's point, where the output relation at Sync's point T is denoted by R[S(T)]. According to the types of the buffered tuples, $\Re$ can modify $\Re_{Sync}$(S) by three different operations as follows: (1) an insert "+" tuple causes $\Re$ to insert a new tuple into $\Re_{Sync}$(S), (2) an update "u" tuple causes $\Re$ to change the values of some attributes of an existing tuple in $\Re_{Sync}$(S), and (3) a delete "-" causes $\Re$ to delete a tuple from $\Re_{Sync}$(S). Notice that update and delete operations can be defined only for relations that have a primary key.

6.2.2 *The Relation-to-Stream Operator* ξ. The operator ξ is responsible for producing a STREAMED (or Incremental) output of a relation. Any synchronized relation $\Re_{Sync}$(S) can be transformed into only one tagged stream that represents the modifications to the relation. Whenever the relation is modified, ξ performs the following: generates delta tuples that represent $\Re_{Sync}$(S)'s modifications since the previous synchronization point and assigns to every generated tuple a timestamp that equals to the modification time. ξ produces the minimum possible set of tuples that represent the delta between two states of the relation. For example, one update tuple is produced for each key value k if k has different attribute values between the two consecutive $\Re_{Sync}$(S) states although k may have been modified by a chain of update operations. For example, in the temperature-monitoring application, the same room may report more than one temperature update in the same synchronization period. However, the set of delta tuples that is generated by ξ at the later synchronization point includes only one update tuple per room that represents the latest temperature update. Notice that the delta between two consecutive states of the relation may be represented in several other ways. For example, ξ can produce the detailed updates chain for a certain key k without performing any accumulation. However, in this paper, we assume that ξ generates the minimum possible set of tuples that can represent the delta between the two states of the relation.

Delta tuples generation: At the i^{th} synchronization time point T_i, ξ generates the delta tuples between $\Re[S(T_{i-1})]$ and $\Re[S(T_i)]$ as follows. For every key value k do the following: (1) If there is a tuple in $\Re[S(T_{i-1})]$ with key k but there is no tuple in $\Re[S(T_i)]$ with key k, then generate a delete tuple for the key k. (2) If there is not a tuple in $\Re[S(T_{i-1})]$ with key k but there is a tuple in $\Re[S(T_i)]$ with key k, then generate an insert tuple for the key k. (3) If there is a tuple with key k in both $\Re[S(T_{i-1})]$ and $\Re[S(T_i)]$ but with different attribute values, then generate an update tuple for the key k.

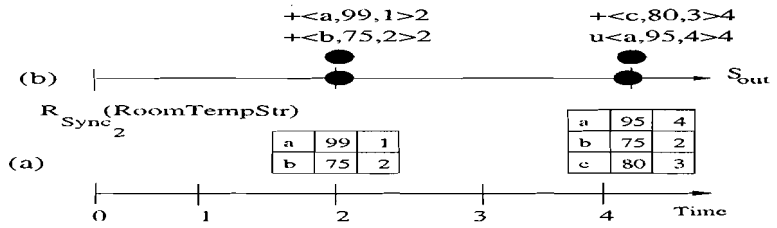


Fig. 5. The Relation-to-Stream Operator.

EXAMPLE 18. Figure 5 gives the mapping from $\Re_{Sync_2}$(RoomTempStr), that is given in Figure 3, to the corresponding stream, S_{out} (i.e., S_{out} = $\xi(\Re_{Sync_2}$(RoomTempStr))). For example, at time 4, ξ produces +<c,80,3>4 and u<a,95,4>4 as the differences since the previous synchronization point, 2. Notice that ξ assigns timestamps to the output stream tuples so that the output stream can be used as input in another continuous query.

6.2.3 *Extended R2R Operators.* The R2R class of operators includes extended versions of the traditional relational operators (e.g., σ, π, $\bowtie$, $\cup$, $\cap$, and -). The semantics of R2R operators in SyncSQL are the same as in the traditional relational algebra. The difference in SyncSQL is that an operator is continuously running to reflect the continuous modifications in the input relations. As with materialized views, the output from an R2R operator is refreshed whenever any of the input relations is modified. For a unary operator (e.g., σ, π), the output relation is modified at the input relation's synchronization points. In other words, the synchronization points (full and partial) for the output are the same as those for the input relation. However, a problem arises in non-unary operators if the input relations have different synchronization points. Notice that operating over relations with different synchronization points is similar to operating over windowed streams with different *slide* values.

For example, consider a binary operator, say O, that has two input synchronized relations, $R_{Sync_1}(S_1)$ and $R_{Sync_2}(S_2)$. The input relation $R_{Sync_1}(S_1)$ is modified at every time point in $Sync_1$ while $R_{Sync_2}(S_2)$ is modified at every point in $Sync_2$. As a result, the output of O is modified at every point $T \in (Sync_1 \cup Sync_2)$. The output of O is interpreted as follows:

—For every time point $T_1 \in (Sync_1 - Sync_2)$, T_1 is a full synchronization point for $R_{Sync_1}(S_1)$ (i.e., at time T_1, $R_{Sync_1}(S_1)$ is up-to-date with S_1). However, the same point T_1 is a partial synchronization point for $R_{Sync_2}(S_2)$ (i.e., at T_1, $R_{Sync_2}(S_2)$ is not up-to-date with S_2). Hence, as a result, T_1 is a partial synchronization point for the output of O because at time T_1, the output of O is not up-to-date with all input streams.

—Similarly, every time point $T_2 \in (Sync_2 - Sync_1)$ is a partial synchronization point for the output of O because it is not up-to-date with all input streams.

—Every time point $T \in (Sync_1 \cap Sync_2)$ is a full synchronization point for the output of O since it is up-to-date with all input streams.

Definition 3. Unary operators. The output of a unary R2R operator Θ over a synchronized relation $\Re_{Sync}(S)$ is another synchronized relation, denoted

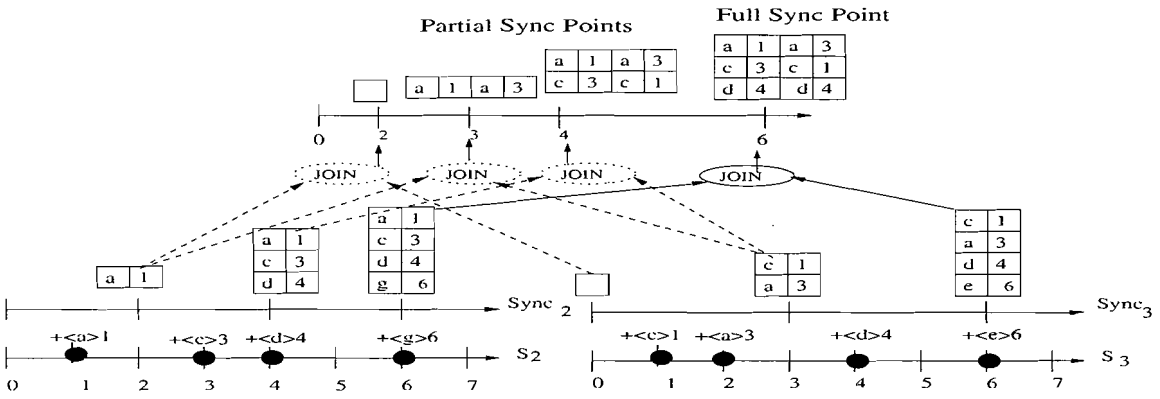


Fig. 6. Joining Relations with Different Synchronization.

by $\Theta(\Re_{Sync}(S))$, such that:
$\forall\, T \in Sync$, *T is a full sync point, and*
$\Theta(\Re_{Sync}(S)) = \Theta(R[S(T)])$, while
$\forall\, T \notin Sync$, *T is a partial sync point, and*
$\Theta(\Re_{Sync}(S)) = \Theta(R[S(\tilde{T})])$
where $\tilde{T} = max\ (t \in Sync\ and\ t < T)$

Definition 4. Binary operators. The output of a binary R2R operator Θ over two synchronized relations $\Re_{Sync_1}(S_1)$ and $\Re_{Sync_2}(S_2)$ is a synchronized relation, denoted by $\Re_{Sync_1}(S_1)\ \Theta\ \Re_{Sync_2}(S_2)$, such that:
(1) $\forall\, T \in Sync_1 \bigcap Sync_2$, *T is a full sync point, and*
$\Re_{Sync_1}(S_1)\ \Theta\ \Re_{Sync_2}(S_2) = R[S_1(T)]\ \Theta\ R[S_2(T)]$,
(2) $\forall\, T \in (Sync_1 - Sync_2)$, *T is a partial sync point, and*
$\Re_{Sync_1}(S_1)\ \Theta\ \Re_{Sync_2}(S_2) = R[S_1(T)]\ \Theta\ R[S_2(\tilde{T})]$,
where $\tilde{T} = max(t \in Sync_2\ and\ t < T)$,
(3) $\forall\, T \in (Sync_2 - Sync_1)$, *T is a partial sync point, and*
$\Re_{Sync_1}(S_1)\ \Theta\ \Re_{Sync_2}(S_2) = R[S_1(\tilde{T})]\ \Theta\ R[S_2(T)]$,
where $\tilde{T} = max(t \in Sync_1\ and\ t < T)$

According to Definition 3, at any time point, say $\grave{T}$, that does not belong to the output synchronization stream, the output synchronized relation from a unary operator reflects the input stream only up to a time point $\grave{\tau}$ where $\grave{\tau} \leq \grave{T}$. Similarly, according to Definition 4, at any time point, say $\grave{T}$, that does not belong to the output synchronization stream, the output from a binary R2R operator reflects one input stream up to time point $\grave{T}$ and reflects the other input stream only up to time $\grave{\tau}$ where $\grave{\tau} \leq \grave{T}$.

EXAMPLE 19. This example demonstrates a join query between two relations, $\Re_{Sync_2}(S_2)$ and $\Re_{Sync_3}(S_3)$, where $Sync_2$ ticks every 2 units while $Sync_3$ ticks every 3 units. The SyncSQL expression is as follows:

```
select STREAMED *
```

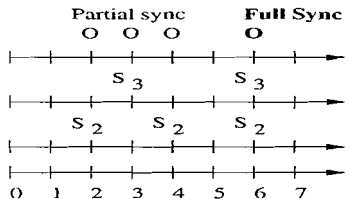


Fig. 7. Join's Partial and Full Synchronization Points.

```
from ℜSync2(S2) R2, ℜSync3(S3) R3
where R2.ID = R3.ID
```

Figure 7 gives the synchronization points for the inputs S_2 and S_3, and the output O. The join output, say O, is refreshed at time points 2, 3, 4, and 6. Figure 6 illustrates the pipeline. The output at 2 is equal to $R[S_2(2)] \bowtie R[S_3(0)]$ and hence 2 is a partial synchronization point since it reflects S_3 only up to time 0. Similarly, 3 is a partial synchronization point since 3 reflects S_2 up to time 2. Also, 4 is a partial synchronization point since 4 reflects S_3 up to time 3. In contrast, 6 is a full synchronization point for the output since 6 reflects *all* input tuples up to time 6. Notice that in practice it makes more sense to use the same synchronization stream with all the join inputs. The synchronization stream that is to be used with the join inputs represent the time points at which the query issuer is interested in the query output.

Distinguishing the full and partial synchronization points is important if the join output is to be used as input to another query. For example, assume a query, say Q_m, that is interested in the result of joining S_2 and S_3 at time 6. Q_m can obtain its desired join result from the output relation at time 6 in Figure 6 because 6 is a full synchronization point and hence, at time 6, $\Re_{Sync_2}(S_2) \bowtie \Re_{Sync_3}(S_3)$ is up-to-date with respect to both S_2 and S_3. In other words, if Q_m is to re-execute the join between S_2 and S_3 at time 6, then Q_m would get the same relation as the output relation at time 6 in Figure 6. On the other hand, assume another query, say Q_n, that is interested in the result of joining S_2 and S_3 at time 3. Unlike Q_m, Q_n cannot obtain its desired output from the output relation at time 3 in Figure 6 because at time 3, $\Re_{Sync_2}(S_2) \bowtie \Re_{Sync_3}(S_3)$ is not completely up-to-date with S_2 and S_3.

6.2.4 *Derived Synchronized Relations.* Based on the previous discussion, the output of an R2R expression over synchronized relations is a derived synchronized relation. The output derived relation has the following logical properties:

—**Data** (or state) that is derived from the input relations' states.

—**Time** that represents the time points at which the derived relation is modified. The derived relation is modified at every time point T if T belongs to the *union* of the input relations' synchronization streams. It is essential to distinguish between the full and partial synchronization points for the derived relation in order to know the time points at which the relation can be used to answer another query. Basically, the time points at which the derived relation are modified are further classified as follows: (1) *Full synchronization points:* a time point T is a full synchronization point for the derived relation only if T is a full synchronization point for all the input relations. (2) *Partial synchronization points:* a time point,

say T, is a partial synchronization point for the derived relation if T is a partial synchronization point for at least one of the input relations.

6.2.5 *Expressing Queries.* In order to express a query over a tagged stream, the SyncSQL expression is constructed as follows. (1) S2R: transform each input stream to the corresponding synchronized relation via an $\Re$ operator using the desired synchronization. (2) R2R: using R2R operators, and in a way similar to traditional SQL, express the query over the synchronized relations. The output of of the R2R pipeline is a derived synchronized relation. (3) R2S: the output synchronized relation is transformed into an incremental output via ξ.

6.3 Equivalences and Relationships

In this section, we introduce preliminary relationships that are required by a query optimizer to enumerate the query plans.

6.3.1 *Containment Relationship among Synchronization Streams.* A synchronization stream, say $\texttt{Sync}_1$, is contained in another synchronization stream, say $\texttt{Sync}_2$, if every time point in $\texttt{Sync}_1$ is also a time point in $\texttt{Sync}_2$ (i.e., $\Re(\texttt{Sync}_1) \subseteq \Re(\texttt{Sync}_2)$). For example, the synchronization stream that is defined over `clockStr` by the predicate "`TimePoint mod 4=0`" is contained in the stream that is defined by the predicate "`TimePoint mod 2=0`".

Proposition 1. $\Re(\texttt{Sync}_1) \subseteq \Re(\texttt{Sync}_2)$ if
$\forall$ I (I $\in$ $\texttt{Sync}_1 \Rightarrow$ I $\in$ $\texttt{Sync}_2$), where I is an insert operation of the form "+<T>T".

6.3.2 *Containment Relationships among Synchronized Relations.* Reasoning about containment relationships between two synchronized relations must consider the two logical properties, state and time, of the relation. For example, consider two synchronized relations, $\Re_{Sync_i}(\texttt{S})$ and $\Re_{Sync_j}(\texttt{S})$, that are defined over the same stream S. Notice that the *states* of $\Re_{Sync_i}(\texttt{S})$ and $\Re_{Sync_j}(\texttt{S})$ may not be equal at every time point if $\texttt{Sync}_i$ and $\texttt{Sync}_j$ are not the same. However, if $\texttt{Sync}_i$ is contained in $\texttt{Sync}_j$, then $\Re_{Sync_i}(\texttt{S})$ is *contained* in $\Re_{Sync_j}(\texttt{S})$. The containment relationship means that every full synchronization time point of $\Re_{Sync_i}(\texttt{S})$ is also a full synchronization point of $\Re_{Sync_j}(\texttt{S})$. The containment relationship is important since $\Re_{Sync_i}(\texttt{S})$ can be computed from $\Re_{Sync_j}(\texttt{S})$ without accessing S. The containment relationship is judged based only on the full synchronization time points of the relation because those are the time points at which the synchronized relation is completely up-to-date with the underlying streams.

Theorem 1. For any stream S, a synchronized relation $\Re_{Sync_i}(S)$ is contained in $\Re_{Sync_j}(S)$ if $\Re(\texttt{Sync}_i) \subseteq \Re(\texttt{Sync}_j)$.

Proof:

(1) Based on Definition 3:
$\Re_{Sync_j}(\texttt{S}) = \texttt{R[S(T)]} \ \forall \ \texttt{T} \in \Re(\texttt{Sync}_j)$;

(2) Given that $\Re(\texttt{Sync}_i) \subseteq \Re(\texttt{Sync}_j)$, then, based on Proposition 1,
$\forall$ T (T $\in$ $\Re(\texttt{Sync}_i) \Rightarrow$ T $\in$ $\Re(\texttt{Sync}_j)$);

(3) From 1 and 2 above,
$\Re_{Sync_j}(\texttt{S}) = \texttt{R[S(T)]} \ \forall \ \texttt{T} \in \Re(\texttt{Sync}_i)$;

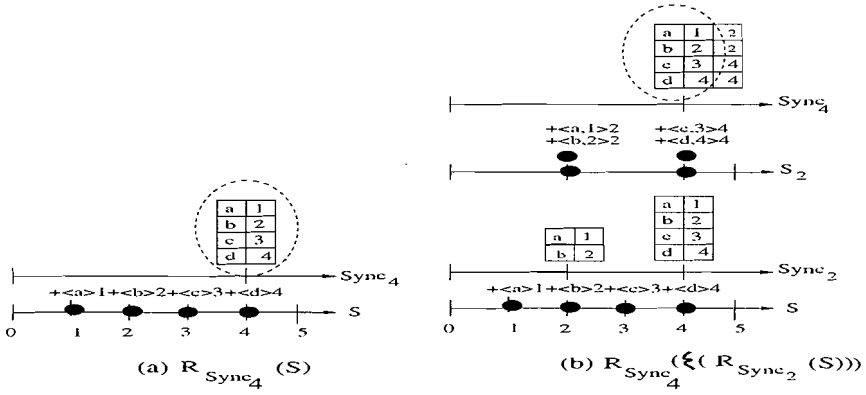


Fig. 8. Relation Containment.

(4) Based on Definition 3,
$\Re_{Sync_i}(S) = R[S(T)] \ \forall \ T \in \Re(Sync_i)$;

(5) From 3 and 4 above:
$\Re_{Sync_i}(S) = \Re_{Sync_j}(S) = R[S(T)]$
$\forall \ T \in \Re(Sync_i)$.

Corollary 1. If $\Re(Sync_i) \subseteq \Re(Sync_j)$, then $\Re_{Sync_i}(S) \subset \Re_{Sync_i}(\xi(\Re_{Sync_j}(S)))$.

Corollary 1 means that $\Re_{Sync_i}(S)$ can be constructed from $\Re_{Sync_j}(S)$ without accessing S. This is done by applying $Sync_i$ over the output stream from $\xi(\Re_{Sync_j}(S))$. Notice that $\Re_{Sync_i}(S)$ does not equal $\Re_{Sync_i}(\xi(\Re_{Sync_j}(S)))$ but $\Re_{Sync_i}(S)$ is contained in $\Re_{Sync_i}(\xi(\Re_{Sync_j}(S)))$. The containment relationship is because $\Re_{Sync_i}(\xi(\Re_{Sync_j}(S)))$ has two timestamp attributes while $\Re_{Sync_i}(S)$ has only one timestamp attribute. The additional timestamp attribute is because in $\Re_{Sync_i}(\xi(\Re_{Sync_j}(S)))$, S is passed by two $\Re$ operators where each $\Re$ operator maps the timestamp attribute of the input stream tuples into the TS attribute of the output relation (as explained in Section 6.2). The containment relationship is further illustrated by Figure 8 where Figure 8a gives the derivation of $\Re_{Sync_4}(S)$ while Figure 8b gives the derivation of $\Re_{Sync_4}(\xi(\Re_{Sync_2}(S)))$.

Proof:

(1) Based on the functionality of the $\Re$ operator, applying $\Re$ with a synchronization stream $Sync_j$ to a stream S does not result in inserting, updating, or deleting any tuples from S. Then, $\Re_{Sync_j}(S)$ exactly represents $S \ \forall \ T \in Sync_j$.

(2) Similarly, based on the functionality of the ξ operator, applying ξ to a relation $\Re_{sync_j}(S)$ does not result in inserting, updating, or deleting any tuples from $\Re_{sync_j}(S)$. Hence, $\xi(\Re_{sync_j}(S))$ exactly represents $\Re_{sync_j}(S)$ $\forall$ point in time.

(3) From 1 and 2 above, $\xi(\Re_{sync_j}(S))$ exactly represents $S \ \forall \ T \in Sync_j$.

(4) For a synchronization stream $Sync_i$ such that $\Re(Sync_i) \subseteq \Re(Sync_j)$, then, $\forall \ T \in Sync_i \Rightarrow T \in Sync_j$.

(5) From 3 and 4 above, $\xi(\Re_{sync_j}(S))$ exactly represents S $\forall \ T \in Sync_i$, hence $\Re_{Sync_i}(S) \subset \Re_{Sync_i}(\xi(\Re_{Sync_j}(S)))$.

EXAMPLE 20. This example illustrates Theorem 1 and Corollary 1. Consider two synchronization streams, $\texttt{Sync}_2$ and $\texttt{Sync}_4$, where $\Re(\texttt{Sync}_4) \subset \Re(\texttt{Sync}_2)$. Figure 8a gives the derivation of $\Re_{Sync_4}(\texttt{S})$ while Figure 8b gives the derivation of $\Re_{Sync_4}(\xi(\Re_{Sync_2}(\texttt{S})))$. Notice that all the full synchronization points for $\Re_{Sync_4}(\texttt{S})$ are also full synchronization points for $\Re_{Sync_2}(\texttt{S})$. Moreover, if only the STREAMED version of $\Re_{Sync_2}(\texttt{S})$ is available (i.e., $\xi(\Re_{Sync_2}(\texttt{S}))$ or $\texttt{S}_2$ in Figure 8b), $\Re_{Sync_4}(\texttt{S})$ can be computed by applying $\texttt{Sync}_4$ over $\texttt{S}_2$ (i.e., $\Re_{Sync_4}(\texttt{S})$ at time 4 is contained in $\Re_{Sync_4}(\xi(\Re_{Sync_2}(\texttt{S})))$ at time 4). A query processor benefits from this containment relationship by sharing the cost of maintaining $\Re_{Sync_2}(\texttt{S})$ among queries that need to maintain $\Re_{Sync_2}(\texttt{S})$ or $\Re_{Sync_4}(\texttt{S})$.

6.3.3 *Commutability between Synchronization and R2R Operators.* R2R operators in a SyncSQL expression are executed over synchronized relations. In this section, we show that the order of applying the synchronization and R2R operators can be switched. The commutability between the synchronization and R2R operators allows executing the query pipeline over finest granularity relations and hence allows sharing the execution among queries that have similar R2R operators but with different synchronization.

Theorem 2. For any *unary* R2R operator Θ, $\forall\ \texttt{T}$ such that T is a full synchronization point of $\Theta(\Re_{Sync}(S))$, T is a full synchronization point of $\Re_{Sync}(\xi(\Theta(\Re(S))))$.

Proof:

(1) From the definition of R2R operators, an R2R operator immediately reflects the changes in the input to the output, then the full synchronization points of $\Theta(\Re_{Sync}(S))$ are the full synchronization points of $\Re_{Sync}(S)$. In other words, the full synchronization points of $\Theta(\Re_{Sync}(S))$ are the time points that belong to the synchronization stream *Sync*.

(2) Since applying the synchronization stream *Sync* is the outermost operation in $\Re_{Sync}(\xi(\Theta(\Re(S))))$, then the full synchronization points of $\Re_{Sync}(\xi(\Theta(\Re(S))))$ are the time points that belongs to the synchronization stream *Sync*.

(3) From 1 and 2 above, the full synchronization points of $\Theta(\Re_{Sync}(S))$ and $\Re_{Sync}(\xi(\Theta(\Re(S))))$ are the same and equals to the time points that belongs to the synchronization stream *Sync*.

Theorem 3. For any *binary* R2R operator Θ, $\forall\ \texttt{T}$ such that T is a full synchronization point of $\Re_{Sync_1}(S_1)\ \Theta\ \Re_{Sync_2}(S_2)$, T is a full synchronization point of $\Re_{Sync_1 \bigcap Sync_2}(\xi(\Re(S_1)\ \Theta\ \Re(S_2)))$.

Proof:

(1) From the definition on non-unary R2R operator, a full synchronization point is a time point at which the output is completely up-to-date with all the input relations. Then, the full synchronization points of $\Re_{Sync_1}(S_1)\ \Theta\ \Re_{Sync_2}(S_2)$ are the time points that are full synchronization points for both $\Re_{Sync_1}(S_1)$ and $\Re_{Sync_2}(S_2)$.

(2) From 1 above, the full synchronization points of $\Re_{Sync_1}(S_1)\ \Theta\ \Re_{Sync_2}(S_2)$ are the time points that belong to the synchronization stream $Sync_1 \bigcap Sync_2$.

(3) Since applying the synchronization stream $Sync_1 \bigcap Sync_2$ is the outermost operation in $\Re_{Sync_1 \bigcap Sync_2}(\xi(\Re(S_1)\ \Theta\ \Re(S_2)))$, then the full synchronization

time points of $\Re_{Sync_1 \bigcap Sync_2}(\xi(\Re(S_1) \ominus \Re(S_2)))$ are the time points that belong to the synchronization stream $Sync_1 \bigcap Sync_2$.

(4) From 2 and 3 above, the full synchronization points of both $\Re_{Sync_1}(S_1) \ominus \Re_{Sync_2}(S_2)$ and $\Re_{Sync_1 \bigcap Sync_2}(\xi(\Re(S_1) \ominus \Re(S_2)))$ are the same and equal the time points that belong to the synchronization stream $Sync_1 \bigcap Sync_2$.

The main idea of Theorems 2 and 3 is that we can pull the synchronization streams out of an R2R operator. Basically, an R2R operator can be executed over the finest granularity relations and produce a finest granularity output. Then, the desired synchronization is applied over the fine granularity output. Notice that Theorems 2 and 3 can also be used in the opposite direction by a query optimizer to push the synchronization inside R2R operators and, hence, reducing the number of operator executions.

7. SYNCSQL QUERY MATCHING

In this section, we introduce a query matching algorithm for SyncSQL expressions. The goal of the algorithm is that, given a SyncSQL query, say Q_i, the algorithm determines whether Q_i (or a part of it) is contained in another view, say Q_j. If such Q_j exists, the algorithm re-writes Q_i in terms of Q_j in a way similar to answering queries using views in traditional databases.

7.1 Peeling SyncSQL Expressions

To reason about containment of SyncSQL expressions, we isolate the synchronization streams out of the expression's data. The containment relationship is then tested in two separate steps: one step to test data containment and another step to test synchronization containment. We term the resulting form of the expressions a "*peeled*" form.

Definition 5. Peeled SyncSQL Expression. The peeled form of a SyncSQL expression is a derived synchronized relation that is defined with: (a) **State**, which is a SQL expression over finest granularity relations, and (b) **Time**, which is a global synchronization stream that specifies the full synchronization points of the expression.

Theorems 2 and 3 are used to transform any SyncSQL expression into the corresponding peeled form. Notice that we can match two expressions only at the full synchronization points because they are the points at which the query answer is up-to-date with *all* the input streams.

Lemma. Any SyncSQL expression has an equivalent peeled form.

EXAMPLE 21. This example derives the peeled form for the SyncSQL expression $Q = \sigma(\Re_{Sync_1}(S_1) \bowtie \Re_{Sync_2}(S_2))$. The derivation is performed in two steps as follows:

-Using Theorem 3, pull the synchronization streams out of the join operator.
$Q = \sigma(\Re_{Sync_1 \bigcap Sync_2}(\xi(\Re(S_1) \bowtie \Re(S_2))))$.

-Using Theorem 2, pull the synchronization stream out of the selection operator.
$Q = \Re_{Sync_1 \bigcap Sync_2}(\xi(\sigma(\Re(S_1) \bowtie \Re(S_2))))$.

The constructed peeled form indicates that Q is equivalent to a synchronized relation with the following: (1) **Data:** $\sigma(\Re(S_1) \bowtie \Re(S_2))$, and (2) **Full synchronization time points:** $\text{Sync}_1 \bigcap \text{Sync}_2$. Notice that the time component gives the full synchronization points for the expression. All other time points that are not full synchronization points are considered to be partial synchronization points.

7.2 Query Matching Algorithm

SyncSQL query matching is similar to view exploitation in materialized views [Goldstein and Larson 2001; Larson and Yang 1985]. However, a matching algorithm for SyncSQL expressions matches the two parts of the peeled forms: state and time.

After introducing the main tools, we now give the high-level steps of the query matching algorithm. The input to the algorithm is a SyncSQL query expression, say Q, and a set of peeled forms for the concurrent queries.

Algorithm SyncSQL-Expression-Matching:

(1) Using Theorems 2 and 3, transform Q to a peeled form by constructing the two components: (1) Q's data, Q^d, and (2) Q's synchronization, Sync_Q;

(2) Match Q^d with data parts of the other input peeled forms using a view matching algorithm from the materialized view literature (e.g., [Goldstein and Larson 2001]). The result of the matching is a peeled form (if any) for a matching expression, say $\tilde{\text{Q}}$, such that $\tilde{\text{Q}}$ consists of a data part $\tilde{\text{Q}}^d$ with synchronization stream $\text{Sync}_{\tilde{\text{Q}}}$.

(3) If such $\tilde{\text{Q}}$ exists, use proposition 1 to check the containment relationship between the synchronization streams Sync_Q and $\text{Sync}_{\tilde{\text{Q}}}$;

(4) If $\Re(Sync_Q) \subseteq \Re(Sync_{\tilde{\text{Q}}})$, then the query Q can be rewritten in terms of $\tilde{\text{Q}}$ as follows. First, rewrite Q^d in terms of $\tilde{\text{Q}}^d$ using the same algorithm used in Step 2 above. In other words, find the function F such that $\text{Q}^d = \text{F}(\tilde{\text{Q}}^d)$.

(5) Apply Q's synchronization Sync_Q to the result of the rewrite in order to get the desired Q's output. In other words, we have $\text{Q} = \Re_{Sync_Q}(\xi(F(\tilde{\text{Q}})))$.

Query matching is used to match an input query against a set of already existing views. On the other hand, if we know the whole set of queries in advance, the peeled forms can be constructed using the greatest common divisor of all synchronization streams instead of the default clock stream.

EXAMPLE 22. This example illustrates the matching of the temperature monitoring query T_4 with the view HotRooms_2 as explained in Example 11. Assume that the input expressions are as follows:

$\text{HotRooms}_2 = \sigma_{Temp\ >\ 80}(\Re_{Sync_2}(RoomTempStr))$

$\text{T}_4 = \sigma_{Temp\ >\ 100}(\Re_{Sync_4}(RoomTempStr))$

The corresponding peeled forms for the two expressions are as follows:

$\text{HotRooms}_2 = \Re_{Sync_2}(\xi(\sigma_{Temp\ >\ 80}(\Re(RoomTempStr))))$

$\text{T}_4 = \Re_{Sync_4}(\xi(\sigma_{Temp\ >\ 100}(\Re(RoomTempStr))))$

By Comparing the two peeled forms we can conclude that: (1)the synchronization stream $Sync_4$ is contained in the synchronization stream $Sync_2$. In other words, $\Re(\text{Sync}_4) \subset \Re(\text{Sync}_2)$, and (2) using a view matching algorithm (e.g., [Goldstein and Larson 2001]) shows that the "Temp > 100" ⇒ "Temp > 80". Then, the algorithm concludes that $\text{T}_4 \subset \text{HotRooms}_2$. Then, the data part of

T_4 can be re-written in terms of $\texttt{HotRooms}_2$ as follows:
$T_4 = \sigma_{Temp > 100}(\xi(\Re(HotRooms_2)))$.
Then, T_4's synchronization is applied to the output of the re-write as follows:
$T_4 = \Re_{sync_4}(\xi(\sigma_{Temp > 100}(\xi(\Re(HotRooms_2)))))$

EXAMPLE 23. This example illustrates query matching. Assume that the stream RoomTempStr has an additional attribute, termed Building, that indicates the building at which the room is located. Consider the following monitoring view over RoomTempStr: "*Group rooms by building and temperature and find the number of rooms in each group. Update the answer every 2 minutes*". The aggregate view is expressed in SyncSQL as follows:

```
CREATE STREAMED VIEW BuildTempGroups AS
SELECT R.Building, R.Temperature, Count(R.RoomID) as cntRooms
FROM ℜ_Sync2(RoomTempStr) R
GROUP BY R.Building, R.Temp
```

Assume further that the following aggregate query, say T_6, is later issued over RoomTempStr: "*Find the number of hot rooms in each building, update the answer every 4 minutes*". T_6 is expressed as follows:

```
SELECT STREAMED R.Building, Count(R.RoomID)
FROM ℜ_Sync4(RoomTempStr) R
WHERE Temperature > 80
GROUP BY R.Building
```

By Comparing T_6 against the view BuildTempGroups we can conclude that: (1) $\Re(\text{Sync}_4) \subset \Re(\text{Sync}_2)$, and (2) using a view matching algorithm (e.g., [Goldstein and Larson 2001]) shows that the view BuildTempGroup contains all the information required to answer T_6. Then, T_6 can be re-expressed as follows:

```
SELECT V.Building, Sum(V.cntRooms)
FROM ℜ_Sync4(BuildTempGroups) V
WHERE V.Temperature > 80
GROUP BY V.Building
```

The Sum aggregate sums the number of rooms in each building. Notice that the view BuildTempGroups can also be used to answer queries over RoomTempStr that requires grouping on the Temperature attribute. Consider, for example, the following query, say T_7: "*Find the number of rooms having the same temperature, update the answer every 2 minutes*". T_7 can be expressed in terms of BuildTempGroups as follows:

```
SELECT V.Temperature, Sum(V.cntRooms)
FROM ℜ_Sync2(BuildTempGroups) V
GROUP BY V.Temperature
```

8. THE NILE-SYNCSQL PROTOTYPE

In this section, we present the design of Nile-SyncSQL, a prototype server to support SyncSQL queries. The Nile-SyncSQL prototype is based on the Nile data stream management system that is developed at Purdue University [Hammad et al. 2004]. Nile is designed to evaluate sliding-window queries over append-only streams. The design of Nile-SyncSQL involves the integration of new concepts

and components into Nile (e.g., tagging, synchronization, and views) along with extending existing components to capture the new concepts (e.g., tagging and synchronization).

8.1 Pipelined Execution of SyncSQL Queries

In Section 4, we showed that continuous queries can be expressed using the same relational operators as snap-shot queries in relational database management systems. Expressing continuous queries using relational algebra has the advantage of being very powerful and can be easily understood. However, since the semantics of relational operators are defined over relations, continuous queries over tagged streams are expressed over the streams' corresponding relations. Notice that a tagged stream represents modifications to a relation with a specific schema. Basically, the logical SyncSQL operators are classified into three classes, stream-to-relation (S2R), relation-to-relation (R2R), and relation-to-stream (R2S). To express a query over a set of input tagged streams, we do the following: (1) transform the streams into corresponding relations using S2R operators, (2) express the query functionality over relations using the relational R2R operators. The output from the R2R expression is a relation, then (3) transform the output relation back to a tagged stream using an R2S operator.

A relational operator (i.e., an R2R operator) takes relations as inputs and produces a relation as output. Logically speaking, when any of the input relations is modified by inserting, updating, or deleting a tuple, the relational operator is "re-executed" over the modified input relation in order to produce a modified output relation. However, usually the modifications in the input relation affect only a small part of the output relation. Hence, the re-execution of the operator involves a lot of redundant computations. Moreover, in case of SyncSQL queries, modifications to the input relations arrive with high rates as tagged streams. Hence, a relational operator (or a relational pipeline) over a tagged stream is to be re-executed with every input stream tuple. As a result, from the implementation and performance point of view, the re-execution approach is not efficient.

The physical implementation of SyncSQL pipelines follows an incremental evaluation approach in order to avoid the re-execution of the pipeline with every input stream tuple. In the incremental evaluation approach, only modifications in the input relations are processed by the query pipeline in order to produce a corresponding set of modifications in the output. In other words, the input tagged stream tuples are processed by the operators in the pipeline in order to produce a tagged stream as output. Basically, an incremental query pipeline is constructed using differential operators instead of the relational operators. Each R2R operator (e.g., σ and $\bowtie$) has a corresponding incremental (or differential) operator (e.g., σ^d and $\bowtie^d$). An incremental operator receives an input stream of tagged tuples and produces another tagged stream as output.

We can say that the physical SyncSQL operators are incremental operators that form a class of stream-to-stream (S2S) operators. Some of the incremental operator may need to keep an internal state to be used to process the input modifications and produce the corresponding modifications in the output. In effect, the functionality of an S2S combines three functions as follows: (1) takes an input modification tuple (i.e., +, u, or -) and applies the modification to the operator's internal state

(if any), (2) performs the relational operator's function over the operator's internal state, then (3) reports the modifications in the internal state as an output tagged stream. We give detailed explanation of the functionality of the S2S operators in Section 0??.

Nile-SyncSQL uses a pipelined queuing model for the incremental evaluation of continuous SyncSQL queries where query pipelines are constructed using incremental operators. Query operators in the pipeline are connected via first-in-first-out queues. An operator, say p, is scheduled once there is at least one input tuple in p's input queue. Upon scheduling, p processes its input and produces output tuples in p's output queue, which is the input queue for the next operator in the pipeline. Tuples that flow in the pipeline are Tagged tuples and can be either insertion (+), update (u), or deletion (-) tuples. The attributes part of a tagged tuple follows the stream's defined schema. An update tuple has an additional part to hold the old attribute values. The old attribute values are first attached by the Tagger operator that is the first operator to produce update tuples in the pipeline (as explained in Section 0??). As the update tuples propagate in the pipeline, the old attributes are processed by the various operators. If an operator is to produce an update tuple as output, the operator is responsible for attaching the old attributes to the output update tuple according to the operator's semantics. An operator gets the old attributes either from the input tuple's old attributes or from the operator's stored state. Old values are needed by the various operators in the pipeline in order to maintain a correct query answer. For example, when an attribute is updated while this attribute is a part of a SUM aggregate. The correct SUM value is constructed by subtracting the old attribute value then adding the new attribute value.

In addition to the incremental operators that implement the query functionality, two new operators are needed to implement the tagging and synchronization principles. The tagging principle is implemented via a Tagger operator. A Tagger operator is needed to transform the input raw streams into tagged streams. Notice that the tagging function is application-dependent and different Tagger operators may need to be implemented. On the other hand, the synchronization principle is implemented via the Synchronizer operator. A synchronizer operator is needed if the query has coarser refresh requirements. Synchronizer is a buffering operator that buffers the input stream tuples and releases them to the query pipeline only at specified synchronization points.

8.1.1 *Example Pipelines.* Figure 9 gives example pipelines for two SyncSQL expressions from Section 4. Figure 9a gives an example from the parking-lot monitoring application while Figure 9b gives an example from the temperature-monitoring application.

The Parking-lot Monitoring Application

Figure 9a gives the pipeline for the parking-lot monitoring view $\texttt{ParkLot}_2$ and P_4 as discussed in Example 10. Figure 9a illustrates that $\texttt{ParkLot}_2$'s pipeline consists of the following operators: (1) A Tagger operator is attached with each one of the input streams. Tagger's output is a stream of "+" tuples since the input streams (i.e., S_1 and S_2) represent append-only relations. (2) A Synchronizer operator is placed on top of each Tagger operator. The Synchronizer's job is to buffer the input tagged tuples and to produce them in the output every 2 minutes when a

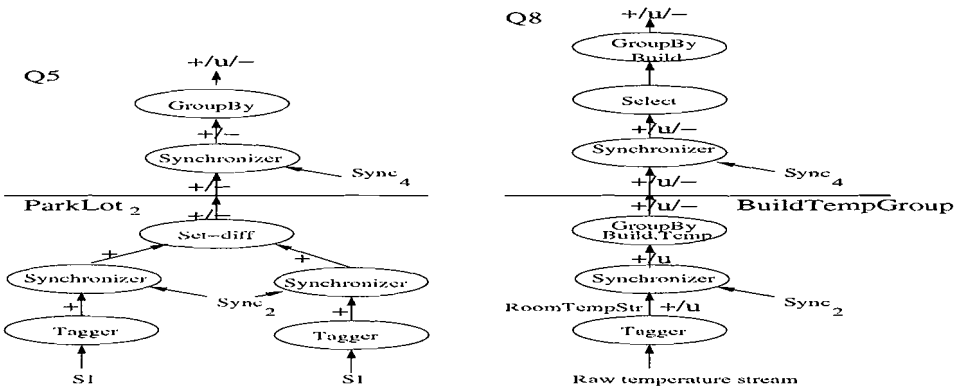


Fig. 9. Examples on SyncSQL Query Pipelines.

synchronization point is received from $\texttt{Sync}_2$. (3) A Set-difference operator that processes the input "+" tuples and produces a tagged stream as output. The Set-difference's output stream represents $\texttt{ParkLot}_2$'s output that includes "+" tuples for vehicles entering the parking lot and "-" tuples for vehicles exiting the parking lot. As discussed in Example 10, Query P_4 is expressed in terms of the $\texttt{ParkLot}_2$ view. As a result, $\texttt{ParkLot}_2$'s output is used as input in $\texttt{P}_4$'s pipeline that consists of two operators, a Synchronizer and a Group-by. $\texttt{P}_4$'s output stream is a tagged stream that includes a "+" tuple for each new group, a "u" tuple for a group whenever the number of vehicles in the group changes, and a "-" tuple whenever a group needs to be deleted because all vehicles in that group exits the lot.

The Temperature Monitoring Application

Figure 9b gives the pipelines for the aggregate view `BuildTempGroups` and $\texttt{T}_6$, as explained in Example 23. Figure 9b illustrates that the pipeline for `BuildTempGroups` consists of the following operators: (1) A Tagger operator that transforms the raw input stream into the tagged stream `RoomTempStr` that includes "+" tuples for newly reported rooms and "u" tuples whenever a room reports a temperature update. The Tagger operator attaches the old temperature values to the "u" tuples because these old values are needed by the following operators in the pipeline. (2) A Synchronizer operator that buffers the input tuples and produces them in the output every 2 minutes, and (3) A Group-by operator that groups the rooms on the `Building` and `Temperature` attributes and maintains the number of rooms in each group. The output of the Group-by operator is a tagged stream that consists of "+", "u", and "-" tuples where a "+" tuple reports the addition of a new group, a "u" tuple reports the change of the number of rooms in a certain group, and a "-" tuple reports the deletion of a certain group due to all rooms in this group having changed their temperatures. `BuildTempGroups`'s output tagged stream is used as input to $\texttt{T}_6$'s pipeline. $\texttt{T}_6$'s pipeline consists of a Synchronizer operator, a Select operator, and a Group-by operator. $\texttt{T}_6$'s Synchronizer buffers the input stream tuples and produces them in the output queue every 4 time units. Select is responsible for selecting only the tuples that represent groups with temperature >

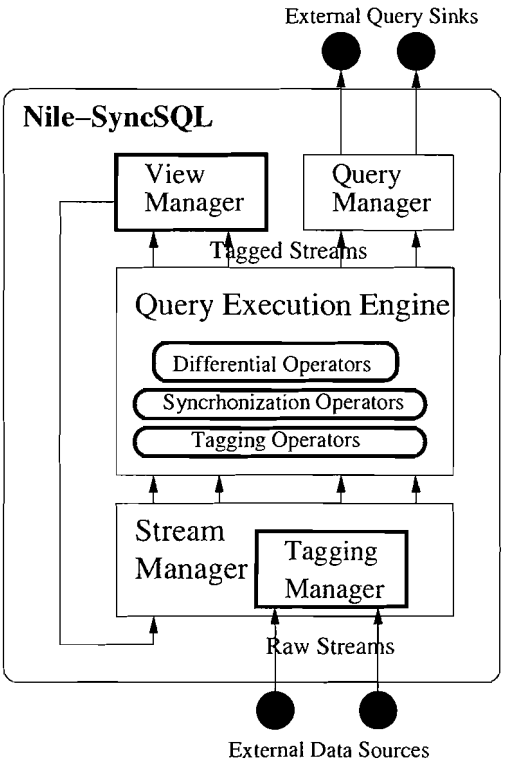


Fig. 10. Nile-SyncSQL Architecture.

80. T_6's Group-by operator groups the input tuples based on the Building attribute.

8.2 Abstract Architecture of Nile-SyncSQL

Nile-SyncSQL is based on Nile, a prototype data stream management system that is developed at Purdue University [Hammad et al. 2004]. Figure 10 gives the architecture of Nile-SyncSQL. The highlighted boxes represent the modules that are added or extended by Nile-SyncSQL.

Nile is a prototype data stream management system to process continuous queries over data streams. Nile adopts a pipelined queuing model for the evaluation of sliding-window queries over append-only streams. Streams are registered into Nile via "CREATE STREAM" statements. The Stream Manager is responsible for maintaining catalog information about the streams and receiving the input stream tuples from the External Data Sources. Queries are issued by External Query Sinks and are registered in the system through the Query Manager. The Query Manager is responsible for constructing the query pipeline while the Query Execution Engine is responsible for the continuous evaluation of the pipelines. The bottom-most operator in any query pipeline is an SSCAN operator that reads the input stream tuples from the Stream Manager. The output from the pipeline's top-most operator is the output of the query and is forwarded to the Query Manager then to the External Query Sink. Nile's query processing engine is designed

to process sliding-window queries over append-only streams. Notice that sliding-window queries form a special class of SyncSQL queries that are characterized by the fact that tuples expire from the window in a First-In-First-Out order.

As illustrated in Figure 10, Nile-SyncSQL extends Nile by adding the following components: the Tagging Manager, Tagging Operators, Synchronization Operators, and the View Manager. At the same time, Nile-SyncSQL requires substantial modifications to Nile's Differential Operators module. The Tagging Manager is responsible implementing the tagging principle. The tagging functions are defined to the Tagging Manager via the "CREATE TAGGED STREAM" statement as discussed in Section 4. Once a tagged stream is used as input to a query, a Tagger operator is added as an interface operator between the input raw stream and the query pipeline. The Tagger operator is responsible for transforming the input raw stream to a tagged stream according to the pre-defined tagging function.

The synchronization principle is implemented via Synchronizer Operators. If a query is interested in coarser refresh periods, a Synchronizer operator is inserted between the input tagged stream and the query pipeline. The Synchronizer operator is responsible for buffering the input stream tuples and passing the tuples into the query pipeline only at the synchronization time points.

The Differential Operators module is responsible for the continuous evaluation of the query expression. Nile's differential operators are designed and optimized to give best performance to the special class of sliding-window queries. Processing SyncSQL queries requires substantial modification to the design of differential operators as follows: (1) generalizing the design of the operators to process tagged streams in which tuples are inserted or deleted in any arbitrary order, and (2) extending the differential operators to support *update* tuples in addition to the *insert* and *delete* tuples.

The View Manager is responsible for defining views and maintaining catalog information about views so that subsequent queries can be answered using the registered views. Once a view is defined, the View Manager is responsible for constructing the view pipeline in a way similar to constructing a query pipeline by the Query Manager. Once constructed, the view pipeline is continuously executed by the Query Processing Engine. The output of the view pipeline is fed to the View Manager to be forwarded back to the Stream Manager so that this output can be used as input to other queries.

8.3 Query Execution Engine

The Query Execution Engine is responsible for the continuous evaluation of the query pipelines. Each query pipeline is constructed from a set of operators that are connected via FIFO queues and each operator runs as a separate thread. When an operator's thread is scheduled, the operator reads a tagged tuple from the input queue, processes the read tuple, and produces a set of tagged tuples in the operator's output queue. In this section, we discuss how the various operators processes the three different types of tuples (i.e., insert, update, and delete tuples).

8.3.1 *The Tagger Operator.* The Tagger operator receives a raw stream as input and produces a tagged stream as output. The functionality of the Tagger operator is application-dependent. For example, in the append-only stream semantics, every

ALGORITHM 1. **Tagging Algorithm for Streams with Primary keys**
Input: t_i : *A raw input stream tuple*
Algorithm

1) Check if a tuple, say t_i^o, is found in state with the same identifer as t_i
2) If t_i^o is found
3) Produce the update tuple: $u< t_i, t_i^o >$
4) Modify t_i^o in the state to be t_i
5) Else
6) Produce a positive tuple $+< t_i >$
7) Insert t_i in the state

Fig. 11. Algorithm of the Tagger Operator.

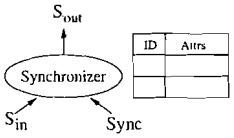


Fig. 12. The Synchronizer Operator.

tuple represents an insertion. Hence, the functionality of the Tagger operator of an append-only stream is as follows: (1) reads a tuple from the input queue, (2) attaches a "+" sign to the tuple, and (3) produces the tagged tuple in the output queue. On the other hand, another Tagger operator is needed for stream that has a primary key to correlate the input tuples. The Tagger operator needs to store one tuple for each key value and attaches tags to the input tuples according to Algorithm 1. The first tuple with a certain key value is produced as a "+" tuple for such object. A following tuple is produced as a "u" tuple over the previous tuple with the same key value. The Tagger operator is responsible for constructing the update tuple by attaching both the old and the new values of the various attributes. Notice that the Tagger's state size has an upper bound that equals to the maximum number of distinct key values.

8.3.2 *The Synchronizer Operator.* The Synchronizer operator is responsible for achieving the synchronization principle. As shown in Figure 12, a Synchronizer operator takes two input streams, S_{in} and Sync, and produces one output stream, S_{out}. S_{in} is the tagged input stream over which the query is expressed while Sync is a synchronization stream that consists of a sequence of time points. Basically, the Synchronizer operator is a buffering operator that is responsible for buffering S_{in}'s tuples and producing them in S_{out} only when a synchronization time point is received from Sync.

For tagged streams, the Synchronizer operator performs summarization on the input tuples. For example, if an object, say O, is inserted then deleted in the same synchronization period, then O is not of interest to the query issuer and hence we can avoid the processing of O's tuples. Hence, the Synchronizer operator digests both O's insert and delete tuples and does not produce them in the output. Moreover,

Table IV. Summarization Rules of the Synchronizer Operator.

New Operation	Previous Operation	Action to Buffer
$+< Attrs >$	$+< Attrs_p >$	This case cannot happen
$+< Attrs >$	u$< Attrs_p, oldAttrs >$	This case cannot happen
$+< Attrs >$	$-< Attrs_p >$	Insert $+< Attrs >$
$+< Attrs >$	Nothing	Insert $+< Attrs >$
u$< Attrs, oldAttrs >$	$+< Attrs_p >$	Delete $+< Attrs_p >$ and Insert $+< Attrs >$
u$< Attrs, oldAttrs >$	u$< Attrs_p, oldAttrs_p >$	Delete u$< Attrs_p, oldAttrs_p >$ and Insert u$< Attrs, oldAttrs_p >$
u$< Attrs, oldAttrs >$	$-< Attrs_p >$	This case cannot happen
u$< Attrs, oldAttrs >$	Nothing	Insert u$< Attrs, oldAttrs >$
$-< Attrs >$	$+< Attrs_p >$	Delete $+< Attrs_p >$
$-< Attrs >$	u$< Attrs_p, oldAttrs_p >$	Delete u$< Attrs_p, oldAttrs_p >$ and Insert $-< Attrs >$
$-< Attrs >$	$-< Attrs_p >$	This case cannot happen
$-< Attrs >$	Nothing	Insert $-< Attrs >$

if another object receives two updates in the same synchronization period, then we can avoid the processing of the earlier update since it is not of interest to the query issuer. Such summarizations reduce the number of tuples processed by the query pipeline without affecting the correctness of the query answer. Table IV lists the possible summarizations that can be performed by the Synchronizer operator. Table IV is interpreted as follows: if an object receives a tuple as indicated in the "`New operation`" column while the same object has already received the tuple as indicated in the "`Previous Operation`" column, then the Synchronizer performs the actions listed in the "`Action to Buffer`" column. We assume correct semantics of the input stream tuples. For example, once a stream tuple with key value k is inserted, no second insertion tuple with key value k is received until after a delete tuple with key k is received. Notice that when summarization is applied, each object can have at most one modification at the end of every synchronization period. The functionality of the Synchronizer operator can be considered as a special Group-by or Aggregate operator that groups the input stream tuple based on the key attribute and produces one output tuple for each group.

The Synchronizer operator works as follows: (1) receives an input tagged tuple from S_{in} and modifies the buffer according to Table IV. (2) Once a `Sync` tuple is received, produced all the tuples in the buffer in S_{out}. Notice that at every synchronization time point, $\Re(S_{in})$ is the same as $\Re(S_{out})$. However, the summarizations that are performed by the Synchronizer operator result in reducing the number of steps that are required to transform a stream to the corresponding relation. Also, notice that for an append-only stream, the summarization process does not reduce the number of output tuples because in append-only streams there are no update or delete tuples. However, synchronizing append-only streams results in producing the query output at regular periods.

8.3.3 *Differential Relational Operators.* Similar to the traditional SQL query pipelines, a SyncSQL query pipeline is constructed using relational operators (e.g., Select, Project, and Group-by). However, SyncSQL pipelines employ incremental evaluation and are constructed using differential versions of the operators. Each relational operator (e.g., Select and Join) has a corresponding differential operator where differential operators differ from traditional operators in that differential operators process modifications (i.e., insert, update, and delete) to relations. Two issues should be distinguished when discussing differential operators: operator *semantics* and operator *implementation*. Operator *semantics* defines the modifications in the operator's output when the operator's input is modified (by inserting, updating or deleting a tuple). On the other hand, operator *implementation* defines the way that the operators realize their semantics. In this section, we discuss the semantics of the various differential operators.

8.3.4 *Incremental Evaluation.* In this section, we use the incremental equations from [Griffin and Libkin 1995] as a guide for discussing the semantics of the various differential operators. Tagged streams are used as inputs and outputs in differential operators. At any time point T, an input stream S can be seen as a relation that is constructed from the input tuples that have arrived before time T. After time T, an input positive tuple s^+ indicates an insertion to S, represented as $(S+s)$, and an expired tuple s^- indicates a deletion from S, represented as $(S-s)$. Two equations are given for every operator, one equation gives the semantics when the input changes by *inserting* a tuple and the other equation gives the semantics when the input changes by *deleting* a tuple. There are no specific equations for the semantics when the input changes by *updating* a tuple since the "update" semantics can be derived as the composition of two operations: "deletion of the old values" and "insertion of the new values". In the following, we assume the duplicate-preserving semantics of the operators. Duplicate-preserving semantics means that duplicate tuples are allowed in the input and output relations of an operator and duplicate tuples are processed independently.

Differential Select $\sigma_p(S)$ and Differential Project $\pi_A(S)$

$\sigma_p(S+s) = \sigma_p(S) + \sigma_p(s) \qquad \sigma_p(S-s) = \sigma_p(S) - \sigma_p(s)$

$\pi_A(S+s) = \pi_A(S) + \pi_A(s) \qquad \pi_A(S-s) = \pi_A(S) - \pi_A(s)$

The incremental equations for Select and Project show that both positive and negative tuples are processed in the same way. The only difference is that positive inputs result in positive outputs and negative inputs result in negative outputs. The equations also show that processing an input tuple does not require access to previous inputs, hence Select and Project are non-stateful operators.

Processing an update tuple in the Select and Project operators is equivalent to the deletion of the old tuple combined with the insertion of the new tuple. In case of the Project operator, an output update tuple is constructed from the old and new tuples after applying the projection. In case of the Select operator, four different outputs can be produced as a result of processing an input "update" tuple as follows:

—If both the old and the new values of the input tuple qualify the selection predicate, then the input update tuple is produced in the output.

—If neither the old nor the new values of the input tuple qualify the selection predicate, no output is produced.

—If only the old values of the input tuple qualifies the selection predicate, then the old input tuple is produced in the output as a "delete" tuple.

—If only the new values of the input tuple qualifies the selection predicate, then the new input tuple is produced in the output as an "insertion" tuple.

Differential Join ($S \bowtie R$)

$(S + s) \bowtie R = (S \bowtie R) + (s \bowtie R) \qquad (S - s) \bowtie R = (S \bowtie R) - (s \bowtie R)$

Join is symmetric which means that processing a tuple is performed in the same way for both input tables. The incremental equations for Join show that, similar to Select, Join processes positive and negative tuples in the same way with the difference in the output sign. Unlike Select, Join is stateful since it accesses previous inputs while processing the newly incoming tuples. The join state can be expressed as two multi-sets, one for each input. Every input tuple t from each input need to be stored in the corresponding input's state even if t does not produce any join outputs. An input tuple need to be stored in the state because it may result in producing a join output with a future tuple from the other input. As a result, the size of each multi-set equals to the number objects in the corresponding input. For example, in the temperature-monitoring application (that is discussed in Section 8.1.1), if the `RoomTempStr` is used as input to Join, the size of `RoomTempStr`'s corresponding state has an upper bound that equals the maximum number of rooms.

Processing an update tuple in Join is performed through the following steps: (1) joins the old tuple with the other table. Assume that the output of this join is a set of tuples, say J_{old}, (2) joins the new tuple with the other table. Assume that the output of this join is a set of tuples, say J_{new}, then (3) produces output tuples as follows:

—Produce *update* tuples that correspond to tuples $\in \mathrm{J}_{old} \bigcap \mathrm{J}_{new}$.

—Produce *delete* tuples that correspond to tuples $\in \mathrm{J}_{old} - \mathrm{J}_{new}$.

—Produce *insert* tuples that correspond to tuples $\in \mathrm{J}_{new} - \mathrm{J}_{old}$.

Differential Set Operations

We consider the duplicate-preserving semantics of the set operations as follows: if stream S has n duplicates of tuple a and stream R has m duplicates of the same tuple a, the union stream $(S \cup R)$ has $(n + m)$ duplicates of a, the intersection stream $(S \cap R)$ has $min(n, m)$ duplicates of a, and the set-difference stream $(S - R)$ has $max(0, n - m)$ duplicates of a.

—**Differential Union ($S \cup R$)**

$(S + s) \cup R = (S \cup R) + s \qquad (S - s) \cup R = (S \cup R) - s$

An input tuple (insert, update, or delete) to the union operator is produced in the output with the same sign. Union is non-stateful since processing an input tuple does not require accessing previous inputs.

—**Differential Intersection**($S \cap R$)

$(S+s) \cap R = (S \cap R) + (s \cap (R-S))$ $\quad$ $(S-s) \cap R = (S \cap R) - (s - (S-R))$

The intersection operator is symmetric. When a tuple s is inserted into stream S, s is produced in the output only if s has duplicates in the set "$R-S$" ("$R-S$" includes the tuples that exist in R and does not exist in S). On the other hand, when a tuple s expires, s should expire from the output only if s has no duplicates in the set "$S-R$". The differential intersection is stateful and the state is expressed as two multi-sets, one for each input. Similar to Join, Intersection stores every input tuple from each input stream in the corresponding state. As a result, the size of the Intersection's state depends on the number of objects in the input stream.

An update tuple is processed by two independent operations: "deletion of the old tuple" and "insertion of the new tuple". The differential intersection operator does not produce update tuples as output. A positive tuple is produced in the output of intersection if the tuple exists in the two input relations. However, whenever a tuple is updated, the new tuple is checked for intersection independent from the old values. If the new tuple is to be produced in the output, then an *insertion* tuple is produced in the output to report the new tuple.

—**Differential Set-Difference** ($S-R$)

Case 1: $(S+s) - R = (S-R) + (s - (R-S))$
Case 2: $(S-s) - R = (S-R) - (s \cap (S-R))$
Case 3: $S - (R+r) = (S-R) - (r \cap (S-R))$
Case 4: $S - (R-r) = (S-R) + (r - (R-S))$

The set-difference operator is asymmetric, which means that processing an input tuple depends on whether the tuple is from S or R. The four cases for the input tuples are handled as follows:

—**Case 1**: An input positive tuple, s^+ from stream S is produced as a positive tuple in the output stream only if s does not exist in the set "$R-S$".

—**Case 2**: An input negative tuple, s^- from stream S is produced in the output stream as a negative tuple only if s exists in the set "$S-R$".

—**Case 3**: An input positive tuple, r^+ from stream R results in producing a negative tuple s^- for a previously produced positive tuple s^+ when s is a duplicate for r and s exists in the set "$S-R$". Notice that the negative tuple s^- is an *invalid* tuple.

—**Case 4**: An input negative tuple, r^- from stream R results in producing a positive tuple s^+ when s is a duplicate of r and s does not exist in the set "$R-S$".

Set-difference is stateful since processing a positive or negative input tuple requires accessing previous inputs. The state is expressed as two multi-sets, one for each input. Similar to Join, Set-difference stores every input tuple from each input stream in the corresponding state. As a result, the size of the Set-difference's state depends on the number of objects in the input stream.

Similar to the differential intersection, an update tuple in the differential set-difference is processed as two independent operations (i.e., deletion of the old

tuple and insertion of the new tuple). Also, differential Set-difference does not produce update tuples as output.

Differential Distinct ϵ

$\epsilon(S + s) = \epsilon(S) + (s - S)$ $\quad$ $\epsilon(S - s) = \epsilon(S) - (s - (S - s))$

The semantics of the distinct operator states that an input positive tuple, s^+, is produced in the output only if s has no duplicates in S (i.e., s exists in the set "$s - S$"). An input negative tuple, s^-, is produced in the output only if s has no duplicates in the set "$S - s$". The differential Distinct is stateful where is the state stores the input stream tuples. Tuples in the state are organized as groups where similar tuples belongs to the same group. The number of groups in the Distinct's state equals to the number of distinct values in the input stream. The size of each group depends on the number of objects that have the corresponding distinct value. Each input object (e.g., room) belongs to at most one distinct group. As a result, the overall size of the Distinct's state equals to the number of objects (e.g., rooms) in the input stream.

Similar to the differential intersection, an update tuple in the differential distinct is processed as two independent operations and no update tuples can be produced as output from the differential distinct.

Differential Aggregates and Group-by

The Group-by operator maps each input stream tuple to a group and produces one output tuple for each non-empty group G. The output tuples have the form $tag < G, Val >$, where G is the group identifier and Val is the group's aggregate value. Notice that the group identifier represents the key attribute for the output stream. The aggregate value Val_i for group G_i is modified whenever the set of G_i's tuples changes, by inserting, updating or deleting a tuple. An update tuple is produced to report the changed group's value. The behavior of Group-by is as follows. When receiving an input positive tuple, say s^+, Group-by maps s to the corresponding group, say G_s, and produces an insertion tuple for G_s, say $< G_s, Val >^+$, if s^+ is the first tuple in G_s. On the other hand, an update tuple is produced for G_s if s^+ is not the first tuple in G_s. Similarly, when a deletion tuple, say s^-, is received, Group-by maps s to the corresponding group G_s, and produces a deletion tuple for G_s if s is the last tuple in the group. On the other hand, an update tuple is produced for G_s if s^- is not the last tuple in G_s. An input update tuple to Group-by may result in producing output tuples for two different groups if the old and new tuples belong to different groups.

Aggregate operator's state: Some aggregate operators (e.g., Sum and Count) do not require storing the input tuples. These aggregates are composable. When receiving a negative tuple, the new aggregate value can be calculated without accessing the previous inputs. Similarly, when receiving an update tuple, the incremental operator uses the old values part to adjust the aggregate value. Other aggregates (e.g., Max) require storing the whole input. In case of Group-by, the state is organized into groups where each input object (e.g., room) belongs to at most one group. The size of each group's state equals to the number of objects that fall in the group. As a result, the overall size of the stateful-aggregate's state equals to number of objects in the input stream.

9. COST ANALYSIS OF SYNCSQL QUERY PIPELINES

In this section, we present a cost model to be adopted by the query optimizer to estimate the cost of a given SyncSQL execution pipeline. The optimizer is a component in the query processing engine that transforms a parsed input query into an efficient query execution plan. The execution plan is then passed to the run-time engine for evaluation. The task of a query optimizer is to find the best execution plan for a given query or a given set of queries. Usually, this goal is accomplished by examining a large space of possible execution plans and comparing these plans according to their "estimated" execution cost. To estimate the cost of an execution plan, the optimizer adopts a cost model that takes several inputs, such as the input arrival pattern, the estimated input size, and the estimated selectivity of the individual operations, and estimates the total execution cost of the query. The different generated plans come from the possibility of using different views to answer the given query.

9.1 CPU Cost Modeling

The execution plan of a SyncSQL query consists of a pipeline of operators that are connected via FIFO queues. As explained in Section 8, operators in the pipeline are continuously running to process the input stream tuples and to produce the corresponding output stream tuples. Traditional database management systems use selectivity information to estimate the cost of a given execution plan up to completion. However, this cost metric does not apply to continuous queries, where the time to complete the query is infinite [Kang et al. 2003]. Hence, the cost model presented in this section finds the cost of executing a given pipeline for a specified period of time. The CPU cost of executing a given plan depends on the following: (1) The number and the organization of operators in the pipeline, (2) the number of tuples processed by each operator, and (3) the CPU cost of processing one tuple in each operator. Basically, the CPU cost of executing a pipeline that consists of n operators for t time units can be estimated as follows:

$$C_{pipeline}(t) = \sum_{i=1}^{n} C_{O_i}(t)$$

where $C_{O_i}(t)$ is the CPU cost of running operator O_i for t time units. $C_{O_i}(t)$ can then be estimated as follows:

$$C_{O_i}(t) = T_i^{in}(t) * c_i$$

where $T_i^{in}(t)$ is the number of input tuples that arrive to O_i during the execution period of t time units and c_i is the CPU cost of processing one tuple in O_i. Notice that c_i is an input parameter that depends on both the system parameters and the implementation. Let $T_i^{out}(t)$ be the number of output tuples from O_i during the execution period. Then $T_i^{in}(t) = T_{i-1}^{out}(t)$. Notice that $T_1^{in}(t)$ is also an input parameter that gives the estimated number of input tuples during t units of time.

Example: Figure 13 gives an execution pipeline that consists of three operators, namely O_1, O_2, and O_3. The CPU cost of executing this pipeline for 60 units of time can be estimated by the following equation:

$$C(60) = T_1^{in}(60) * c_1 + T_2^{in}(60) * c_2 + T_3^{in}(60) * c_3$$

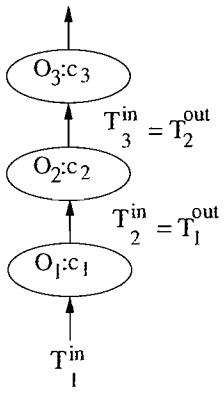


Fig. 13. Estimating the Cost of SyncSQL Execution Pipelines.

9.2 Estimating the Output Cardinality of the Various Operators

In this section, we present a model to estimate the output cardinality of an operator (i.e., T^{out}). The output cardinality of an operator depends on the number of input tuple (i.e., T^{in}) and on the operator functionality. In the following, we discuss the relationship between T^{in} and T^{out} for the various types of operators. The number of input tuples for the bottom-most operator, denoted as $T_1^{in}(t)$, is an input parameter. Notice that if the bottom-most operator is a non-unary operator, then $T_1^{in}(t)$ is the summation of all the input tuples from all the input streams. Then, $T_1^{in}(t)$ can be used to estimate $T_1^{out}(t)$, which in turn equals to the number of input tuples of the above operator in the pipeline (i.e., $T_2^{in}(t) = T_1^{out}(t)$). Generally, we trace the pipeline bottom-up using $T_{i-1}^{in}(t)$ to estimate $T_i^{in}(t)$.

9.2.1 *The Synchronizer Operator.* The synchronizer operator is responsible for buffering the input stream tuples and for producing output tuples only at the synchronization time points. If the input stream is append-only, then the number of output tuples from the synchronizer operator equals to the number of input tuples (i.e., $T^{out}(t) = T^{in}(t)$). However, if the input stream includes modification tuples (i.e., update and delete tuples), then the synchronizer operator accumulates the input tuples according to Table IV, where a new update from an object, say K_j, overwrites the previous update to K_j. As a result, the number of output tuples from the synchronizer operator is less than the number of input tuples (i.e., $T^{out}(t)$ $<= \mathrm{T}^{in}(t)$). $T^{out}(t)$ depends on the update pattern of the various objects. Notice that different objects may have different update patterns. For example, consider an input stream that represents updates of K objects (e.g., K rooms). Then, T^{out} can be estimated as:

$$T^{out} = \sum T^{out}(t)|K_j$$

where $T^{out}(t)|K_j$ is the number of output tuples in t time units (i.e., $T^{out}(t)$) due to K_j and $T^{out}(t)|K_j$ depends on the relationship between K_j's update pattern and the frequency of synchronization points. Remember that the synchronizer operator produces an update tuple for K_j at the i^{th} synchronization point, say x_i, if there is at least one K_j's update that is reported between x_{i-1} and x_i. In order to find the number of output tuples from the synchronizer operator we need to study both

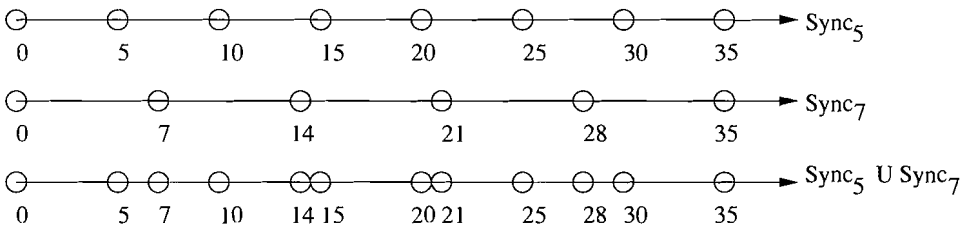


Fig. 14. Union of Two Synchronization Streams.

(1) the object update patterns, and (2) the patten of synchronization points.

Object Update Patterns

An object update pattern represents the pattern that is followed by an object to report updates for its values. A single stream may represent updates for a large number of objects. However, the different objects may have different update patterns. As a result, $T^{out}(t)|K_j$ is estimated for each K_j independently. The output of the synchronizer operator is then estimated as follows:

$T^{out}(t) = \sum_{\forall j} T^{out}(t)|K_j$.

Some of the common object update patterns are as follows:

—**Uniform update pattern:** In this pattern, an object, say K_j, reports an update every y_j time units, where y_j's value differs from one object to another. Notice that the values of y_j for the different objects are provided as input parameters. For example, consider a temperature-monitoring application that monitors the temperature for a certain number of rooms. A common input to such application can be as follows: a certain room, say r_1, reports a temperature update every 5 seconds, however a different room, say r_2, reports an update every 3 seconds, and so on.

—**Poisson update pattern:** In this pattern, the arrival rate of K_j's updates follows a Poisson arrival model [Ahrens and Dieter 1974]. In other words, on average, K_j reports λ updates per unit time and the interarrival time between two consecutive updates follows the exponential distribution with mean $1/\lambda$. The probability a K_j's update will be produced at a synchronization point, say x_i, equals to the probability that there is at least one K_j arrival in the interval between x_i and the preceding synchronization point x_{i-1}. From the properties of the Poisson process, the probability that there is at least one arrival in an interval of length l equals to $1 - e^{\lambda l}$.

Patterns of Synchronization Points

We can classify the synchronization streams according to the pattern of synchronization points into two classes as follows.

—**Uniform synchronization:** In this case, the synchronization points are equally spaced. For example, the synchronization stream $Sync_2$ that is used in the temperature-monitoring queries in Section 4 has a synchronization point every 2 time units.

—**Non-uniform Synchronization:** In this case, the distance between two con-

secutive points is not fixed. Such non-uniform synchronization can be the result of constructing synchronization streams as the union or intersection of other uniform synchronization streams. For example, consider Figure 14 that gives two synchronization streams, namely $Sync_5$ that has a synchronization point every 5 time units and $Sync_7$ that has a synchronization point every 7 time units. Figure 14 also gives the synchronization stream $Sync_5 \cup Sync_7$ that has a synchronization point every 5 or 7 time units. Notice that the synchronization intervals in $Sync_5 \cup Sync_7$ are of different lengths. For the sake of analysis, we assume that for every synchronization stream, the synchronization points are repeated every cycle of a given length. For example, the synchronization points in $Sync_5 \cup Sync_7$ (Figure 14) are repeated in a cycle every 35 time units.

Estimating $T^{out}(t)|K_j$

The function that estimates $T^{out}(t)|K_j$ depends on both K_j's update pattern and the pattern of synchronization points as follows.

—**Uniform update and synchronization patterns:** Assume that K_j reports an update every y_j time units and that there is a synchronization time point every x time units. Then, there are two cases as follows:

(1) $y_j <= x$: In this case, there is at least one K_j's arrival between any two synchronization points. As a result, at every synchronization point, an update tuple is produced for K_j. Hence, $T^{out}(t)|K_j$ equals to the number of synchronization points or $T^{out}(t)|K_j = \lceil t/x \rceil$.

(2) $y_j > x$: In this case, there is at least one synchronization point between every two subsequent K_j's updates. As a result, every K_j update will be produced in the output and no accumulation of tuples takes place in the synchronizer operator. Hence, $T^{out}(t)|K_j$ equals to the number of input tuples from K_j or $T^{out}(t)|K_j = \lceil t/y_j \rceil$.

—**Uniform update and non-uniform synchronization patterns:** Assume that K_j sends an update every y_j time units and that the synchronization time points are repeated in a cycle of length L. In this case, in order to estimate $T^{out}(t)|K_j$, we need to perform a check at every synchronization point in a cycle of length LCM(L,y_j). For each synchronization point in the cycle, say x_i, we check whether there is at least one K_j's update between x_{i-1} and x_i or not. In other words, an update is produced for K_j at x_i if and only if $\lfloor x_i/y_j \rfloor - \lfloor x_{i-1}/y_j \rfloor > 0$.

For example, consider the synchronization stream $Sync_5 \cup Sync_7$ that is given in Figure 14. Assume that an object K_j reports an update every 3 time units. In order to find $T^{out}(t)|K_j$, we need to analyze the synchronization points in a cycle of length $LCM(35, 3) = 105$. Figure 15 illustrates the result of the analysis where the black circles indicate the synchronization points that produce an update for K_j. Figure 15 illustrates that there are 33 synchronization points in each cycle of length 105 time units and only 27 synchronization points produce update tuples due to K_j. Then, for an execution period of length 630 time units, the cycle of length 105 is repeated 630/105 times or 6 times. As a result, $T^{out}|K_j(630) = 27 * 6$.

—**Poisson update and non-uniform synchronization patterns:** Assume that K_j sends updates according to a Poisson arrival pattern with an average rate of

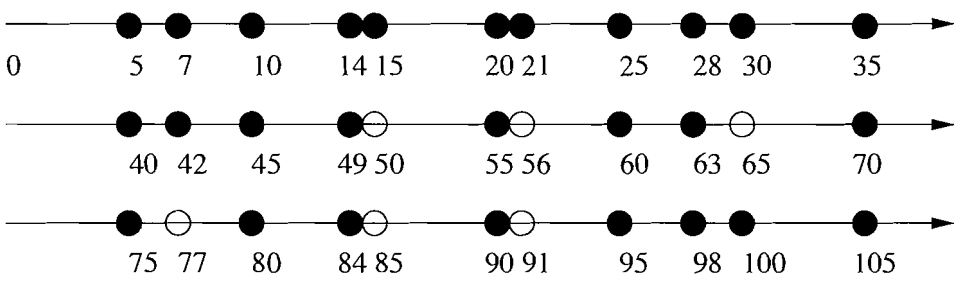


Fig. 15. Uniform Update and Non-uniform Synchronization.

λ_j time units and that the synchronization time points are repeated in a cycle of length L. In this case, we need to analyze the synchronization points in a cycle of length L such that for every synchronization point, say x_i, we find the probability that there is at least one K_j arrival in the interval between x_i and the preceding synchronization point x_{i-1}. Figure 14 illustrates that there are 11 intervals in each cycle of the synchronization stream $Sync_5 \cup Sync_7$. There are 3 intervals of length 5 and 2 intervals for each one of the following lengths: 4, 3, 2, and 1. By applying the formula $1 - e^{-\lambda\, l}$ on the different intervals, we find that, on the average, 7 synchronization points in each cycle will result in producing updates for K_j.

—**Poisson update and uniform synchronization patterns:** Assume that K_j sends updates according to a Poisson arrival pattern with an average rate of λ_j time units and that the synchronization time points are uniform every y_j time units. In this case, all the synchronization intervals are of length y_j. Hence, the number of output tuples at every synchronization point is estimated by the formula $1 - e^{-\lambda\, y_j}$.

9.2.2 *The Tagger, Union, and Aggregate Operators.* The Tagger, Union, and Aggregate operators are alike in that they do not perform any filtration or summarization of the input tuples. In other words, every input tuple to any of these three operators results in producing one tuple in the output stream, i.e., $T^{out}(t) = T^{in}(t)$. However, these operators differ in the function that transforms an input tuple to the corresponding output tuple. For example, the Tagger operator attaches a tag to the input tuple and produces the tagged tuple in the output. However, the aggregate operator produces an update tuple with the new aggregate value after aggregating the input tuple's value.

9.2.3 *The Select Operator.* The Select operator applies a predicate on the input tuples and produces tuples in the output as explained in Section 8. The number of output tuples from Select depends on the predicate selectivity, say f, where f is an input parameter. Generally, $T^{out}(t) = f * T^{in}(t)$. Notice that an input tuple to Select can produce at most one output tuple. In other words, $0 \leq T^{out} \leq T^{in}$.

9.2.4 *The Join Operator.* Similar to Select, the number of output tuples from the Join operator depends on a predefined join selectivity, say f. However, unlike Select, an input tuple to Join may result in producing more than one output tuples depending on the join multiplicity. Assume that the sizes of the Join inputs in t

time units are estimated by $T_1^{in}(t)$ and $T_2^{in}(t)$. Then, the number of output tuples from Join is estimated by $T^{out}(t) = f * T_1^{in}(t) * T_2^{in}(t)$.

9.2.5 *The Group-by Operator.* Every input tuple (e.g., insert, update, or delete tuple) to the Group-By operator modifies the aggregate value of at least one group. As a result, every input tuple to Group-By results in producing at least one output tuple. However, an input *update* tuple may result in producing two output tuples if the old and new values belong to two different groups. Generally, in Group-by, $T^{in}(t) \leq T^{out}(t) \leq 2T^{in}(t)$. Notice that if the grouping function is on the key attribute (or on an attribute that is dependent on the key attribute), then both the old and new values of an update tuple belong to the same group and only one output tuple is produced in response to this input tuple. On the other hand, if the grouping function involves an updateable attribute, then each one of the old and new values of an update tuple may belong to a different group and there may be two output tuples produced in response to this input tuple.

9.2.6 *The Set-difference, Intersect, and Distinct Operators.* The Set-difference, Intersect, and Distinct operators are similar in that each input tuple results in producing zero, one, or two output tuples. Generally, for these three operators, we have $0 \leq T^{out}(t) \leq 2T^{in}(t)$. The result of processing an input tuple has three cases: (1) no output tuples, for example, when an input *insert* tuple to the Distinct operator belongs to an already existing distinct group, (2) one output tuple, for example, when an input *insert* tuple to the Distinct operator represents the first tuple in its corresponding distinct group, or (3) two output tuples, for example, when an input *update* tuple to Distinct in which each one of the old and new values belongs to a different group. As a result, the new part may result in producing an insert tuple for the corresponding group while the old part may result in producing a delete tuple for a different group.

9.3 Example

In this section, we give an example to illustrate how the proposed cost model is used by the query optimizer to choose the best execution plan for a given set of concurrent overlapping queries. As discussed in Section 6, given a set of overlapping concurrent queries, we can define a view that represents the overlapping part of the queries. Then, we use the output of the view as input to the various queries. As a result, the execution of the view is shared among the various queries. However, views should be used only if they result in enhancing the performance of the query processing engine.

Consider the following two queries from the temperature-monitoring application:

—Q_1: Continuously monitor rooms with temperature greater than 95. Report modifications in the answer every 2 time units.

—Q_2: Continuously monitor rooms with temperature less than 80, report modifications in the answer every 4 time units.

Q_1 and Q_2 are queries over the `RoomTempStr` stream that reports updates to the temperatures of the various rooms. Q_1 uses the synchronization stream $Sync_2$ that has a synchronization point every 2 time units, while Q_2 uses the synchronization

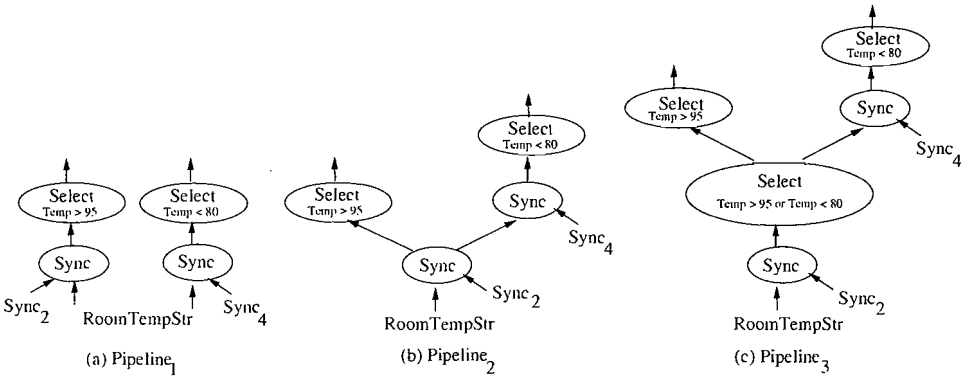


Fig. 16. Plan Enumerations for a Set of Two Aggregate Queries.

stream $Sync_4$ that has a synchronization point every 4 time units. Figure 16 gives three possible execution scenarios for the two queries as follows:

—Pipeline$_1$ in Figure 16a gives the independent execution of the two queries.

—Pipeline$_2$ in Figure 16b gives a shared execution plan in which a synchronizer operator is shared between the two queries. The shared synchronizer operator uses $Sync_2$ as the synchronization stream since $Sync_2$ represents $Sync_2 \cup Sync_4$.

—Pipeline$_3$ in Figure 16c gives another shared execution plan in which the shared plan consists of two operators: a Synchronizer operator and a Select operator. Notice that the shared Synchronizer uses $Sync_2$ as the synchronization stream and the shared Select predicate is union of the two disjoint predicates (i.e., Temp > 90 or Temp < 80).

Assume that there are 2000 different rooms and that each room reports updates in a uniform pattern every 0.1 time units. Assume further that the CPU costs of processing one tuple in the Synchronizer and the Select operators are c_1 and c_2, respectively. Let NS be the number of tuples that are processed by the Synchronizer operators and NL be the number of tuples that are processed by the Select operators. As a result, the CPU cost for running a pipeline for 65 time units can be estimated as $C_{pipeline}(65) = NS * c_1 + NL * c_2$.

A note about Pipeline$_3$: In order to find the number of output tuples from the upper synchronizer operator (with synchronization stream $Sync_4$), we need to know the update patterns of the various objects in the input stream. However, the update patterns of the objects in input stream are not known since this stream represents the output from a Select operator. We need to find the number of output tuples from the following expression: $\Re_{Sync_4}(\xi(\sigma(\Re_{Sync_2}(RoomTempStr))))$. As discussed in Section 6, since the order of the select and synchronization is commutable, then:
$\Re_{Sync_4}(\xi(\sigma(Sync_2(RoomTempStr)))) = \Re_{Sync_4}(\xi(\Re_{Sync_2}(\sigma(RoomTempStr))))$
Moreover, since $Sync_4 \in Sync_2$, then we have:
$\Re_{Sync_4}(\xi(\Re_{Sync_2}(\sigma(RoomTempStr)))) = \Re_{Sync_4}(\xi(\sigma(RoomTempStr)))$
Again, by switching the order of synchronization and selection we have:
$\Re_{Sync_4}(\xi(\sigma(RoomTempStr))) = \sigma(\Re_{Sync_4}(RoomTempStr))$.

Finding the number of output tuples from the pipeline $\sigma(\Re_{Sync_4}(RoomTempStr))$ is straightforward since the update patterns of objects in `RoomTempStr` are known and hence the number of output tuples from the synchronizer operator can be estimated. Notice that the switch between the Synchronizer and Select operator is used here to illustrate how to estimate the number of output tuples from the pipeline that corresponds to the expression $\Re_{Sync_4}(\xi(\sigma(\Re_{Sync_2}(RoomTempStr))))$. However, at the query execution time, the order of executing the Synchronizer and Select operators depends on the query parameters.

The cost for the three pipelines can be estimated as follows.

—Since every room reports an update every 0.1 time units, then the number of input `RoomTempStr` tuples in 65 time units = (65/0.1)*2000 = 1300000 tuples.

—In 65 time units, there are 65/2 or 32 synchronization points in $Sync_2$ and there are 65/4 or 16 synchronization points in $Sync_4$. Each Synchronizer produces one output tuple for each room at each synchronization point. Hence, the number of output tuples from the Synchronizer operator with the $Sync_2$ synchronization stream are 32*2000 = 64000 tuples. Similarly, the number of output tuples from the Synchronizer operator with the $Sync_4$ synchronization stream are 16*2000 = 32000 tuples.

—For $Pipeline_1$, `RoomTempStr`'s input tuples are processed twice by the the two Synchronizer operators, then $NS = 1300000 * 2 = 2600000$. The two Select operators process the output tuples from the two Synchronizer operators. Then, $NL = 64000 + 32000 = 96000$. As a result, $C_{Pipeline_1}(65) = 2600000 * c_1 + 96000 * c_2$

—For $Pipeline_2$, `RoomTempStr`'s tuples are processed by the lower Synchronizer operator (with $Sync_2$ synchronization stream) and the output tuples from the lower Synchronizer operator are processed by the upper synchronizer operator (with $Sync_4$ synchronization stream). Then, $NS = 1300000 + 64000 = 1364000$. Similar to $Pipeline_1$, the two Select operators process the output tuples from the two Synchronizer operators. Then, $NL = 96000$. As a result, $C_{pipeline_2}(65) = 1364000 * c_1 + 96000 * c_2$

—For $Pipeline_3$, `RoomTempStr`'s tuples are processed by the lower Synchronizer operator (with synchronization stream $Sync_2$). The upper Synchronizer operator processes the output tuples from the shared Select operator. Assume that the selectivity of the shared Select operator is 0.5. Hence, the number of output tuples from the shared Select operator = 0.5 * 64000 = 32000. Then, $NS = 1300000 + 32000 = 1332000$. The number of input tuples to the shared Select operator is 64000 and the number of input tuples to the Select operator with predicate (Temp > 95) is 0.5*64000 = 32000. As explained in the note about $Pipeline_3$ above, the number of input tuples to the Select operator with predicate (Temp < 80) is 32000 *0.5 = 16000. Then $NL = 64000+32000+16000 = 112000$. As a result, $C_{pipeline_3}(65) = 1332000 * c_1 + 112000 * c_2$

By analyzing the previous equations we conclude the following: (1) $Pipeline_2$ has less CPU cost than that of $Pipeline_1$ because $Pipeline_2$ requires a fewer number of synchronization operations. (2) The preference between $Pipeline_2$ and $Pipeline_3$ depends on the values of c_1 and c_2.

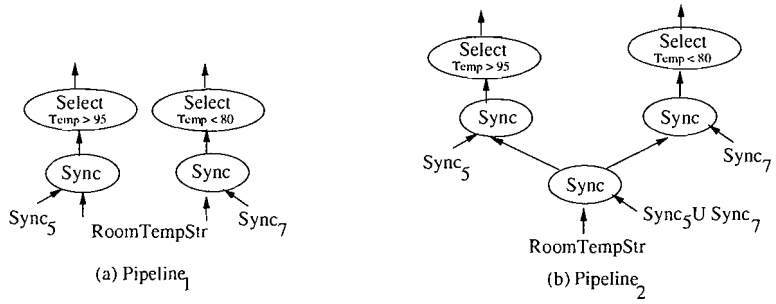


Fig. 17. Effect of Input Parameters.

9.3.1 *Effect of the Input Parameters.* Assume that we repeat the previous example but with a different set of input parameters. Assume that Q_1 is that the answer needs to be modified every 5 time units, and that Q_2 is such that the answer needs to be modified every 7 time units. In the case of independent execution (Pipeline $_1$ in Figure 16a), Q_1's synchronizer operator uses the $Sync_5$ synchronization stream that is given in Figure 14. Similarly, Q_2's pipeline uses the $Sync_7$ synchronization stream. Moreover, the shared synchronizer operator uses the $Sync_5 \cup Sync_7$ synchronization stream. Figure 17 gives two possible execution scenarios as follows:

—Pipeline$_1$ in Figure 17a gives the independent execution of the two queries.

—Pipeline$_2$ in Figure 17b gives a shared execution plan in which a synchronizer operator is shared between the two queries. The shared synchronizer operator uses $Sync_5 \cup Sync_7$ as the synchronization stream.

Assume that there are 2000 different rooms and each room report updates in a uniform pattern every 3 time units. Using similar analysis to that of Figure 16, the CPU cost for running the two pipelines in Figure 17 for 650 time units can be estimated by the following equations:

—$C_{Pipeline_1}(650) = 866666 * c_1 + 445714 * c_2$

—$C_{Pipeline_2}(650) = 1101904 * c_1 + 445714 * c_2$

The analysis of these equations shows that Pipeline$_1$ gives better performance than Pipeline$_2$ because Pipeline$_1$ uses less synchronization operations. In other words, in this scenario, the shared execution of queries using views worsen the performance of the system. The conclusion is that the decision on whether to share the execution among queries using views or not depends on the input parameters. Hence, the query optimizer uses the input parameters along with the proposed cost model to choose the most effective query execution plan. In Section 10, we gave experimental results to validate the proposed cost model.

10. EXPERIMENTAL EVALUATION OF NILE-SYNCSQL

In this section, we give an experimental evaluation of the Nile-SyncSQL prototype. The goal of the experimental evaluation is to (1) analyze the factors that affect the performance of SyncSQL queries, and (2) demonstrate the effectiveness of supporting views in data stream management systems. We run experiments to evaluate

and compare various possible realizations for the newly introduced concepts (e.g., tagging and synchronization). Another set of experimental results shows that views can be used as an efficient means for the shared execution of continuous queries.

10.1 Experimental Setup

The Nile-SyncSQL prototype is implemented on Intel Pentium 4 CPU 2.4 GHz with 512 MB RAM running Windows XP. A continuous query is evaluated via a pipeline of operators where each operator in the pipeline runs as an independent thread. The threads communicate with each others via FIFO queues. A producer-consumer locking mechanism is implemented to control the queue access in a way that a queue is accessed by at most one thread at a time. Operators' threads are scheduled using a round-robin scheduling where each operator runs for a fixed amount of time to consume tuples from the operator's input queue. Once the input queue of the operator is exhausted or the operator's time slot is finished, the next operator is scheduled.

10.1.1 *Workload Queries.* We use queries from the temperature-monitoring application (that is discussed in Section 4) to evaluate the performance of Nile-SyncSQL. The temperature-monitoring application allows us to study the performance of the tagging principle since a tagging function is defined to transform the input streams into tagged streams by correlating the input tuples based on the primary key attribute RoomID. Moreover, the temperature-monitoring application allows us to test the performance while three different types of tuples (i.e., insert, update and delete) flow in the query pipeline. Two input streams are generated, namely TemperatureSource and HumiditySource. The TemperatureSource stream is a stream that reports the various rooms' temperature and has a schema of three attributes as follows: (RoomID, Building, Temperature), where RoomID is an integer attribute that gives the room identifier, Building is an integer attribute that represents the building in which the room resides, and Temperature is an integer attribute that gives the temperature reading of this room. Similarly, Humiditysource is a stream that reports the various rooms' humidity and has a schema of three attributes as follows: (RoomID, Building, Humidity). The RoomID is the key attribute for both the TemperatureSource and HumiditySource streams and an input stream tuple is an update over the previous tuple with the same RoomID value. A tagging transformation is defined to transforms TemperatureSource and HumditiySource streams into the tagged streams RoomTempStr and RoomHumStr, respectively. We use the following operators to construct various query pipelines over TemperatureSource and HumiditySource: Tagger, Synchronizer, Select, Project, Join, Group-by, Aggregate. Then, we run the various query pipelines and collect measurements (e.g., execution time, throughput, and memory consumption) and use the collected measurements to evaluate the performance of Nile-SyncSQL.

10.1.2 *Data Generation.* We use randomly generated synthetic data in our experiments. To generate the TemperatureSource stream, we specify the number of distinct identifiers (i.e., number of rooms) and the number of buildings, where the rooms are evenly distributed among buildings. Then, we specify the arrival rate for the stream where the arrival rate is defined as the number of stream tuples to be received by the system in one second. The inter-arrival time between two data

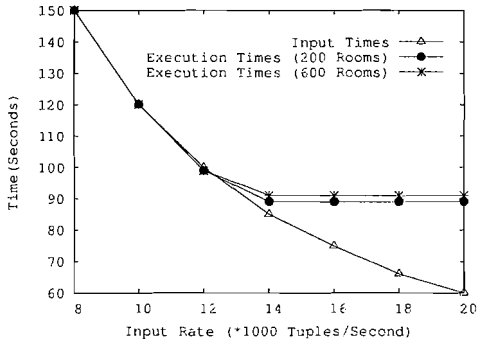


Fig. 18. Effect of Arrival Rate.

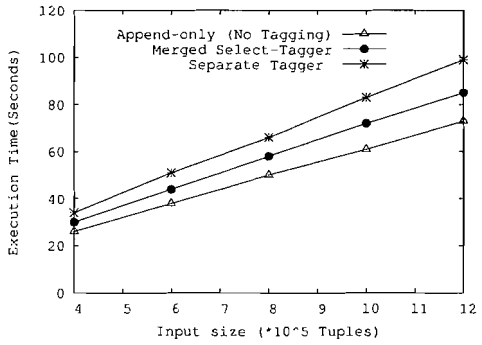


Fig. 19. Cost of the Tagging Operation.

items follows the exponential distribution with mean λ tuples/second. The arrival rate of the input streams is changed by varying the parameter λ of the exponential distribution. We generate the stream tuples such that the arrival rate is evenly distributed among the rooms (if not mentioned otherwise). For example, in a stream that reports readings from 200 rooms with arrival rate 20000 Tuples/Second, each room reports its temperature 100 times per second. The temperature readings are varied from 73 to 100 and are extracted from a real temperature sensor readings from the SensorNet projects at CMU [Pervasive Infrastructure Sensor Networks].

10.2 Performance of the Tagger Operator

In this section, we analyze the factors that affect the performance of the Tagger operator and propose optimizations to minimize the overhead of tagging. We first run an experiments to measure the Tagger's throughput, where the throughput is defined as the maximum number of tuples that can be processed by the Tagger operator per time unit. Notice that Tagger's throughput depends on the complexity of the tagging transformation. We run a query pipeline that consists of only a Tagger operator where `TemperatureSource` is used as input. We run the experiment several times while varying the number of distinct room identifier in `TemperatureSource`. The pipeline works as follows: Tagger reads a tuple from `TemperatureSource`, uses the tuple to maintain the state, attaches the corresponding tag, and produces the tagged tuple in the output.

Figure 18 gives the effect of the input arrival rate on the query execution time. We measure the time that is taken by the pipeline to process 1.2 million input tuples while varying the arrival rate from 6000 to 20000 Tuples/Second. The graphs in Figure 18 give the input and execution times. The "Input Times" graph illustrates that for the same number of input tuples, the input time decreases as the arrival rate increases. However, the "Execution Times" graphs illustrate that the execution time initially decreases with the increase in the arrival rate then saturates when the arrival rates reaches 14000 Tuples/second. Two "Execution Times" graphs are given to illustrate the execution time when the updates in `TemperatureSource` are sent by 200 and 600 rooms. Before saturation (i.e., for arrival rates less than 14000 Tuples/Second) the execution time is the same as the input time, which means

ALGORITHM 2. **The Merged Select-Tagger Operator**
Input: t_i : *A raw input stream tuple*
Output: t_o : *A tagged stream tuple*
Algorithm

1) Apply the selection predicate on t_i
2) If t_i *does not qualify the predicate*
3) Check if a tuple t_i^o *is found in state with the same identifer as* t_i
4) If t_i^o *is found*
5) Delete t_i^o *from state*
6) Output the delete tuple $-< t_i^o >$
7) Else
8) ignore t_i
9) Else if t_i *qualifies the predicate*
10) Check if a tuple t_i^o *is found in state with the same identifer as* t_i
11) If t_i^o *is found*
12) Produce the update tuple: $u< t_i, t_i^o >$
13) Modify t_i^o *in the state to be* t_i
14) Else
15) Produce a positive tuple $+< t_i >$
16) Insert t_i *in the state*

Fig. 20. Algorithm of the Merged Select-Tagger Operator.

that the system is not overloaded and that the input tuples are processed as fast as they arrive. At saturation, the execution time is fixed at 90 seconds even if the arrival rate is larger than 14000 Tuples/Second. The conclusion is that the maximal throughput of the Tagger operator is around 14000 Tuples/Second. The graphs in Figure 18 illustrate also that the Tagger's throughput is the same when the number of rooms is 200 or 600. The throughput is independent from the number of distinct key values because both streams have the same number of input tuples and each input tuple takes the same amount of time to be processed independent from how many tuples are processed for the same room identifier.

10.2.1 *Merged Select-Tagger operator.* In the temperature-monitoring application, the Tagger operator maintains information for all the distinct room identifiers in order to correlate the input tuples. At the same time, a query may be interested in only a small number of rooms (e.g., by having a selection predicate on the `RoomID` attribute). As a result, the overhead of the Tagger operator can be reduced if the Tagger is aware of the query's selection predicate. In order to minimize the tagging overhead, we propose to merge the tagging functionality with the Select operator. The merged `Select-Tagger` operator receives the raw input stream tuples, evaluates the selection predicate, and assigns appropriate tags to the output tuples. Algorithm 2 gives the pseudo code for the algorithm of the merged `Select-Tagger` operator.

We use the $\texttt{HotRooms}_1$ view from Example 6 (in Section 4) to evaluate the performance of the merged `Select-Tagger` operator. The straightforward $\texttt{HotRooms}_1$'s pipeline consists of two operators: Tagger and Select. The required projection predicate is implemented inside the `Select` operator because in `Nile-SyncSQL`,

both the selection and projection predicates are implemented inside Select. TemperatureSource is the input stream to $\texttt{HotRooms}_1$'s pipeline and the output stream is a stream that represents the rooms with temperature > 80. The Tagger operator maintains a state that contains one entry for each distinct room identifier and produces insert and update tuples for the various rooms. Notice that an update tuple contains two sets of attributes that represent the old and new values for that tuple. The output from the Tagger operator is then used as input input to Select. When processing an update tuple, Select applies the selection and projection predicates twice, once on the old values and once on the new values.

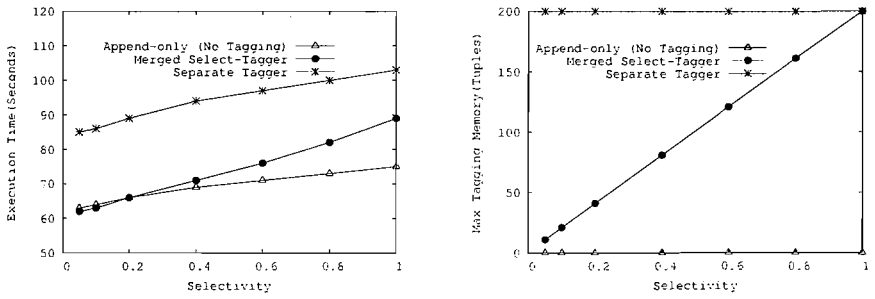
If we apply the merged Select-Tagger optimization, the optimized $\texttt{HotRooms}_1$'s pipeline will consist of one operator, namely the merged Select-Tagger operator. The merged operator improves both the memory and CPU consumption of the query as follows:

—**Memory:** only rooms that qualify the selection predicate are stored in the state. Memory savings can be considerable when the query employs highly selective predicates.

—**CPU:** The merged Select-Tagger operator reduces the CPU cost of the query pipeline due to the following: (1) avoid updating the state by tuples that correspond to rooms that do not qualify the selection predicate, and (2) avoid the re-execution of select and project predicates on the old part of an update tuple by getting the result of processing the old part from the stored tagging state.

Figure 19 gives the execution times that are taken to process different input sizes to $\texttt{HotRooms}_1$'s view. The number of rooms is set to 200 and the arrival rate is fixed to 20000 Tuples/Second while the input size is varied from 400000 to 1.2 million tuples. The three graphs in Figures 19 compare three cases: (1) a pipeline of two operators, namely Tagger and Select, (2) a pipeline with one operator, namely the merged Select-Tagger operator, and (3) a pipeline with only Select operator. Notice that the pipeline in case (3) does not give the desired query semantics since the input tuples are not correlated based on the RoomID attribute. However, we include this case to quantify the tagging overhead. Figure 19 gives the throughput of the three different pipelines as follows: (1) 12K Tuples/second (2) 14K Tuples/second, and (3) 16K Tuples/second. These values of throughput indicate that the merged Select-Tagger operator results in 15% increase in the system throughput compared with the separate Tagger operator. The increase in system throughput is due to the reduction in the number of state modifications, and in the number of selection and projection evaluations. Moreover, Figure 19 illustrates that the overhead of tagging reduces the throughput by 10% reduction in contrast to the no-tagging pipeline (i.e., Pipeline 3). The conclusion is that although processing update tuples doubles the number of selection and projection evaluations, it does not double the execution time because it does not double the communication cost nor the cost of constructing the output tuples.

10.2.2 *Effect of Selectivity.* In this section, we study the effect of selectivity on the performance of the merged Select-Tagger operator. We divide this study into two sections as follows: key selectivity and non-key selectivity. Key-selectivity experiments are based on queries in which the selection predicate is defined on



(a) Execution Time (b) Memory

Fig. 21. Effect of Key-selectivity on Tagger's Performance.

the key attribute of the input stream (e.g., RoomID). In contrast, the non-key selectivity experiments are based on queries in which the selection predicate is on a non-key attribute (e.g., Temperature). The key and non-key selectivities differ in their effect on the query performance. In the case of key selectivity, once an object (i.e., room) qualifies the predicate, the object continues to qualify the predicate for as long as the query is running. As a result, once a qualified object is inserted in the `Select-Tagger`'s state, the object will not be deleted and the size of the Tagger's state will be fixed during the query runtime. On the other hand, for non-key selectivity, an object may fluctuate between qualifying and disqualifying the query predicate. As a result, the size of the `Select-Tagger`'s state will vary during the query runtime.

Effect of Key Selectivity

Figure 21 gives the effect of key selectivity on the tagging cost. This experiment is performed for a pipeline that is similar to that of $\texttt{HotRooms}_1$'s view pipeline while changing the selection predicate. The graphs in Figure 21 compares the performance of the same three pipelines in Figure 19. Figure 21a gives the effect of selectivity on the query execution time while Figure 21b gives the effect of the selectivity on the memory consumption. The input size in this experiment is 1.2 million tuples. The selection predicate is on `RoomID` attribute and selectivity is varied from 0 to 1. Figure 21a illustrates that the merged `Select-Tagger` operator achieves 30% improvement in the query execution time if compared with the separate Tagger operator. The reason is that the separate Tagger operator performs a lot of unneeded state maintenance operations since all room identifiers are stored and used to update the Tagger's state. Figure 21a also illustrates that for low selectivity values (less than 0.5) the tagging overhead is almost zero and the merged `Select-Tagger` pipeline has the same execution time as the No-Tagging pipeline. The tagging overhead started to appear from selectivity values larger than 0.5. The merged operator performs slightly worse than the append-only performance because of the state maintenance operations.

Figure 21b gives the effect of the selectivity on the memory usage. For the

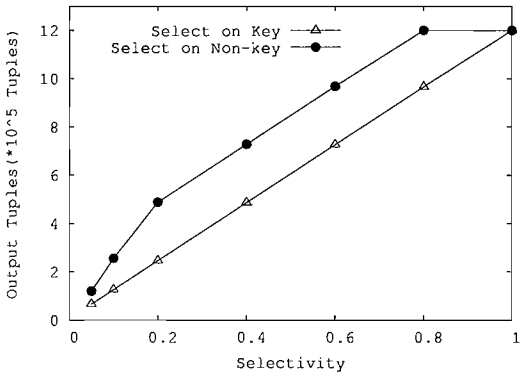


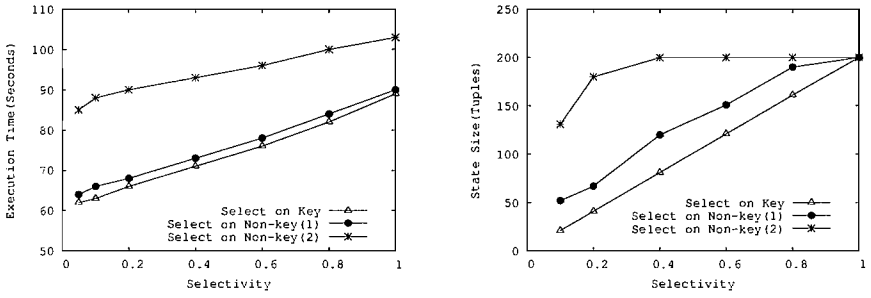
Fig. 22. Key- vs. Non-key Selectivity.

separate Tagger operator pipeline, the memory requirement is independent from the selectivity and equals to 200 tuples (i.e, the maximum number of rooms) because Tagger stores all rooms even the rooms that do not qualify the query predicate. For the merged `Select-Tagger` operator the state size is proportional to the selectivity because only rooms that qualify the selection predicate are stored in the state. In other words, the merged `Select-Tagger` operator has the minimum possible memory requirement for the correct query evaluation. For the append-only stream semantics there is no tagging and hence the memory requirements equal to zero. However, in this case, the output stream does not convey the required semantics.

Effect of Non-key Selectivity

In this section, we illustrate the difference between key- and non-key- selectivity. We use a pipeline that consists of one merged `Select-Tagger` operator. Figure 22 compares the number of output tuples from the pipeline when the selection predicate is on a key or on a non-key attribute. The number of input tuples is 1.2 million tuples and the selectivity is varied from 0 to 1. A selectivity of value 0.2 for example means that, out of the 1.2 million input tuples, there are 240000 tuples that qualify the selection predicate. When the selection predicate is on a key-attribute, the number of output tuples exactly matches the selectivity factor. However, the number of output tuples may exceed the selectivity factor if the selection predicate is on a non-key attribute. The extra tuples are basically negative tuples that are produced due to the fact that some rooms may be deleted from the output several times depending on the object update pattern. Notice that the number of output tuples gives an indication to the query execution time. The execution time of a query increases as the number of tuples in the pipeline increases since every tuple needs to be constructed, communicated between the operators, and to be processed by the various operators in the pipeline.

Figure 23 compares the performance of the `Select-Tagger` pipeline in the case of the key and non-key selectivities. Moreover, Figure 23 illustrates the effect of the input data distribution on the performance. We run the same query with a non-key selectivity predicate on two different input streams. The two input streams differ in the update pattern of each room. For example, assume that a certain room, say R_i,



(a) Execution Time (b) Memory

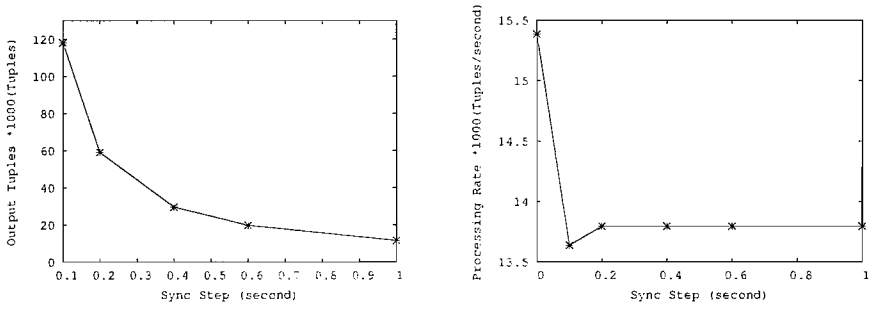
Fig. 23. Effect of Non-key Selectivity on Tagger's Performance.

reports 4 temperature readings in the following order: 89, 87, 79, and 78. Assume further that the selection predicate is as follows: Temperature > 80. As a result, Room R_i will result in producing three output tuples as follows: +, u, -. However, assume that in another distribution, Room R_i reports the same four readings but in a different order as follows: 89, 79, 87, 78. In this latter distribution, Room R_i will result in producing four output tuples as follows: +, -, +, -. Notice that although Room R_i has the same number of qualified readings (i.e., 0.5 selectivity), the number of output tuples depends on the distribution of the qualified tuples.

Figure 23a gives the effect of data distribution on the execution time while Figure 23b gives the effect of data distribution on memory requirement. The input size is 1.2 million tuples and the selectivity is varied from 0 to 1. The graphs illustrate that for the same selectivity value, a query with a non-key predicate may encounter more processing time and memory than a query with a key predicate. Moreover, the execution time of the non-key predicate varies from one data distribution to another. For example, for non-key distribution 2, objects fluctuate in and out of the query boundary more than that in distribution 1. As a result, distribution 2 causes more deletions and insertions into the state and hence more processing of negative tuples.

Figure 23b illustrates that state size of the merged `Select-Tagger` operator may reach the maximum number of distinct key values which is the same as the separate Tagger pipeline. This is because it can happen that all the rooms satisfy the query predicate at the same time. However, the CPU cost of the merged operator is always better than that of the separate operator due to the savings in the number of selections and projections.

10.2.3 *Discussion.* We can summarize the results from the tagging experiments as follows: (1) the CPU overhead of the tagging principle is minimized by the merged `Select-Tagger` operator and is negligible in most of the queries, (2) the memory overhead of the tagging principle is limited and has an upper-bound that equals to the number of key identifiers that satisfy the query predicate, (3) the selectivity factor affects both the CPU and memory consumption. Notice that the



(a) Tuples (b) Execution Time

Fig. 24. Performance of the Synchronizer Operator.

memory overhead of tagging is similar in spirit to the memory required by the sliding-window operators (e.g., SEQ-WINDOW in [Arasu et al. 2006] and W-EXPIRE in [Ghanem et al. 2007]). Moreover, processing a tuple twice (old and new) is also similar to processing a tuple twice in sliding-window queries (in sliding-window queries, a tuple is processed twice: once as new and once as expired).

10.3 Performance of the Synchronizer Operator

In this section, we analyze the factors that affect the performance of the Synchronizer operator and study the effect of synchronization on query performance. We first run an experiment to study the effect of the synchronization period on the query execution time. We run a query pipeline that consists of a Tagger and a Synchronizer operators where TemperatureSource is used as input to the Tagger and the Tagger's output is used as input to the Synchronizer. The pipeline works as follows: Tagger reads a tuple from TemperatureSource, attaches the corresponding tag, and produces the tagged tuple in the output. Then, Synchronizer reads a tagged tuple, maintains the buffer by performing the corresponding summarizations as explained in Section 8, and produces the buffered tuples as output when a tuple is received from the synchronization stream.

Figure 24 gives the effect of the synchronization period on the number of output tuples and on the query execution time. The number of distinct room identifiers in TemperatureSource is set to 200 and the input rate is fixed to 16000 Tuples/Second while the synchronization period is varied 0.1 to 1 second. Figure 24a gives the number of output tuples from the Synchronizer operator while varying the synchronization step. As the synchronization period increases, the number of output tuples decreases. The reason for this decrease in the number of output tuples is that, in a bigger synchronization period, a larger number of update tuples are digested (i.e., summarized) by the Synchronizer operator, and hence fewer number of tuples are processed by the upper Tagger operator. At every synchronization step, at most one output tuple can be produced for each room. For example, when the synchronization period is 0.1 second, one tuple is produced for each room every 0.1

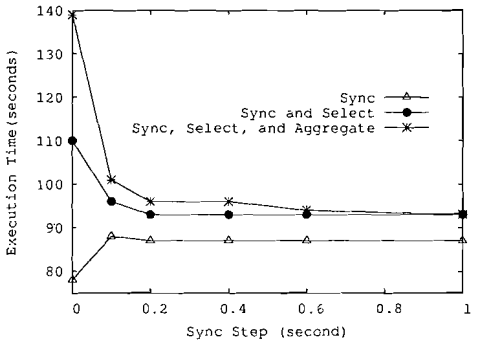


Fig. 25. Effect of Synchronization on Query Execution Time.

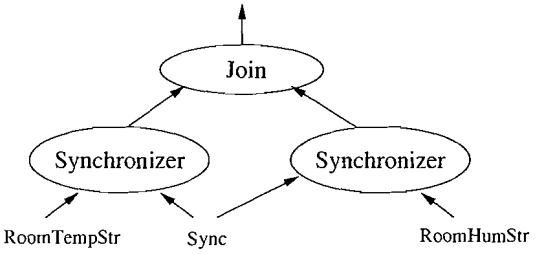


Fig. 26. Pipeline of the Join Query.

second. However, when the synchronization step is 0.2, one tuple is produced for each room every 0.2 second. As a result, the number of output tuples for synchronization step 0.2 is almost half the number of tuples for synchronization step 0.1. The number of output tuples from the Synchronizer operator gives an indication for the required query resources since these output tuples are processed by the upper operators in the pipeline.

Figure 24b gives the processing rate of the query pipeline while varying the synchronization step. The processing rate is defined as the maximum number of tuples that can be processed by the pipeline in one time unit. The graph in Figure 24b illustrates that the pipeline can process up to 15400 Tuples/Second when the synchronization step is set to 0 (i.e., no buffering is needed). However, once buffering starts (at synchronization period 0.1 second), the processing rate of the pipeline drops to 13600 Tuples/Second. The decrease in the processing rate is due to the cost of updating the buffer in the Synchronizer operator. Figure 24b also illustrates that the processing rate of the Synchronizer operator is fixed to 13600 Tuples/Second independent of the synchronization step. The processing rate is fixed because the same number of input tuples are processed by the Synchronizer operator independent of the synchronization step. Therefore, the synchronization step affects only the number of output tuples.

10.3.1 *Effect of the Synchronization Period on Query Performance.* Figure 25 gives the effect of synchronization on query execution time. Figure 25 gives the execution times that are needed to process 1.2 million for the following three different pipelines: (1) a pipeline that has only the Synchronizer operator, (2) a pipeline that has two operators, Synchronizer and Select, and (3)a pipeline that has three operators: Synchronizer, Select, and Group-By operator. The same tagged stream (that corresponds to the `TemperatureSource` stream) is used as input for the three pipelines and the input rate is fixed to 20000 Tuples/Second.

Figure 25 illustrates that the query execution time decreases as the synchronization period increases. The reason for the decrease in execution time is the decrease in the number of tuples processed by the query pipeline. Moreover, the percentage

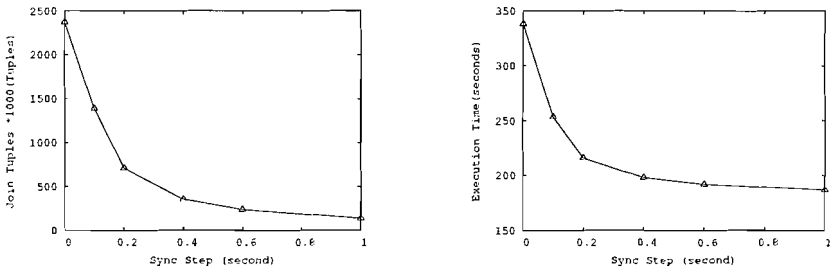
of reduction in execution time depends on the complexity of the query. For example, when the synchronization step increases, the execution time drops by 25% in the case of Pipeline 3 while the execution time for Pipeline 2 drops by only 10%. The execution time for Pipeline 1 includes the time for processing the input tuples by the Synchronizer operator only. As a result, the execution time for Pipeline 1 increases when the synchronization step increases because of the cost of summarizing and buffering the input tuples by the Synchronizer operator. Notice that once synchronization starts (i.e., at sync step 0.1), only 200 tuples (one for each room) are produced by the Synchronizer operator to be processed by the pipeline in every synchronization step. The small number of room identifiers leaves the pipeline not overloaded and the input tuples are processed as fast as they arrive.

In the following experiment, we study the effect of the synchronization period on the query execution time of a join query. Consider a join query between the temperature and humidity streams, namely `RoomTempStr` and `RoomHumStr`, that is interested in the continuous monitoring of the temperature and humidity readings of the various rooms. Assume that the query is interested in reporting the modifications in the answer every 2 minutes. Such join query is expressed in SyncSQL as follows:

```
select STREAMED R1.RoomID, R1.Temperature, R2.Humidity
from ℜSync2(RoomTempStr) R1, ℜSync2(RoomHumStr) R2
where R1.RoomID = R2.RoomID
```

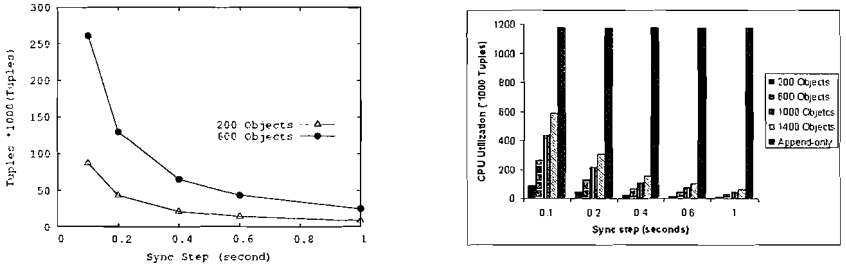
Figure 26 gives the query pipeline that consists of a join operator and two Synchronizer operators. The number of distinct room identifiers is set to 1400 and the input rate for each stream is 20000 Tuples/Second (i.e., total input rate to the pipeline is 40000 Tuples/Second). Figure 27 gives the number of join tuples and the execution times taken to process a total of 2.4 million input tuples (i.e., 1.2 million tuples from each input stream). Figure 27a illustrates that the number of tuples processed by the join operator decreases as the synchronization step increases. As a result, the CPU cost of the pipeline decreases as the synchronization step increases. Figure 27b confirms the previous results by showing that the query execution time also decreases as the synchronization step increases. Notice that the synchronization principle reduces the execution time of the query pipeline at the cost of giving lower resolution query answers.

10.3.2 *Effect of the Number of Distinct Key Values.* The number of distinct key values (e.g., number of rooms in the temperature-monitoring application streams) affects the number of output tuples from the Synchronizer operator. The reason is that at every synchronization point, Synchronizer produces at most one output tuple for every distinct key value. As a result, for the same number of input tuples and the same synchronization step, the number of output tuples from the Synchronizer increases as the number of distinct key values increases. Figure 28a gives a comparison of the number of output tuples from the Synchronizer operator when the number of objects in the underlying stream is 200 and 600 while varying the synchronization step from 0.1 to 1 second. More tuples are produced from the Synchronizer operator when the underlying stream has 600 room identifiers. Notice that the number of tuples that flow in the pipeline is an important measurement since it gives an indication of the query execution time. Figure 28b gives a sum-



(a) Tuples (b) Execution Time

Fig. 27. Effect of Synchronization on Join Performance.



(a) (b)

Fig. 28. Effect of Number of Objects.

mary of the relationships among the number of key values, the synchronization step, and the number of tuples. From the figure, we notice that: (1) For the same synchronization step, as the number of distinct key values increases, the number of tuples in the pipeline increases (2) For the same number of key values, as the synchronization step increases, the number of tuples flowing in the pipeline decreases. Notice that in append-only streams, each tuple in the stream has a distinct key value, hence the synchronization step has no effect on the number of tuples flowing in the pipeline. However, synchronizing an append-only stream has the effect of refreshing the query answer at regular time intervals independent from the arrival pattern of the input tuples. A synchronizer operator over an append-only stream collects the input stream tuples and produces them at regular intervals.

10.4 Aggregate Queries and Pre-synchronization

Consider the following aggregate query from the temperature-monitoring application: " Find the number of hot rooms in each building, report modifications in the answer every 2 time units". This aggregate query is expressed in SyncSQL in two

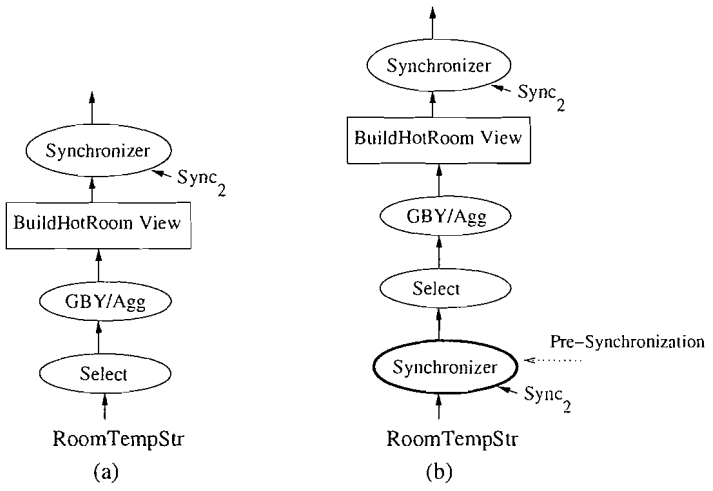


Fig. 29. Pipeline of the BuildHotRoom View.

steps as follows. First, we need to define a view that finds the number of hot rooms in each building as follows:

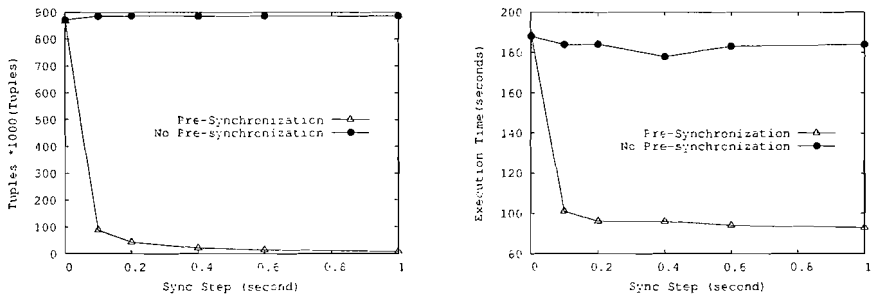
```
CREATE STREAMED VIEW BuildHotRooms AS
SELECT R.Building, Count(R.RoomID) as cntRooms
FROM ℜ(RoomTempStr) R
WHERE Temperature > 85
GROUP BY R.Building
```

Notice that the `Building` attribute represents the key attribute for the output stream from the `BuildHotRooms` view. An update tuple is produced in the output stream from the `BuildHotRooms` view whenever a room enters or exits the query range. Notice that the same building receives several updates if the building has more than one hot room. Notice also that the query issuer asks to be notified by the modifications in each building "once" every two time units. In order to get the desired output, we apply the desired synchronization (i.e., every 2 time units) on `BuildHotRooms`'s output as follows:

```
SELECT V.Building, V.cntRooms
FROM ℜ_Sync2(BuildHotRooms) V
```

The output stream from the last query includes at most one update tuple for each building on every synchronization time point, hence achieving the desired query semantics. Figure 29a gives the query pipeline that is used to execute the `BuildHotRooms` view and the subsequent query. Notice that the Synchronizer's state size equals the maximum number of buildings because `R.Building` represents the key field for the output stream tuples `BuildHotRooms`.

The Pre-synchronization Optimization: In the room temperature monitoring application, each room sends updates to its temperature more than once every unit time as explained in the data generation procedure in Section 10.1. As a result, the same room may result in producing several update tuples in `BuildHotRooms`'s output stream for the corresponding building. The update tuples that are pro-



(a) Tuples (b) Execution Time

Fig. 30. Effect of Pre-synchronization.

duced from the same building are summarized by the Synchronizer operator to produce a single output for each building in every synchronization step. Notice that all the updates tuples that result from the same room belong to the same building, and hence are summarized by the Synchronizer operator as well. This observation highlights the possibility of pre-summarizing the updates for each single room and include only one update from each room in the final building summarization. The pre-summarization can be achieved by performing a pre-synchronization on the `RoomTempStr` stream before being processed by `BuildHotRooms`'s pipeline. Figure 29b gives the optimized query pipeline by adding an additional Synchronizer operator at the bottom of the pipeline. The added Synchronizer operator results in that each room has at most one update tuple to be processed by the aggregate operator in each synchronization period. As a result, each room affects `BuildHotRooms`'s output once in every synchronization period. Pre-synchronization results also in reducing the CPU time taken by the aggregate operator since less tuples flow in the pipeline. Pre-synchronization is similar in spirit to eager aggregation that is performed to enhance the performance of aggregate queries in traditional databases [Yan and Larson 1995].

Figure 30 gives a comparison of the performance of the two pipelines that are given in Figure 29, when processing an input stream of 1.2 million tuples. The input stream has 200 distinct key values and the arrival rate is 20000 Tuples/Second. Figure 30a gives the number of tuples processed by the aggregate operator while varying the synchronization step from 0 to 1. All the input tuples are processed by the aggregate operator when no pre-synchronization is performed. However, the number of tuples is reduced significantly when pre-synchronization is applied since the bottom Synchronizer operator digests many input tuples. Figure 30b gives the query execution time that is proportional to the number of tuples processed by the query pipeline. Pre-synchronization reduces the execution time by about 50%. The reduction in the execution time is due to the reduction in the number of tuples processed by the operators in the pipeline.

10.5 Experimental Verification of the Cost Model

In this section, we experimentally verify the accuracy of the proposed cost model to estimate the CPU cost of SyncSQL execution plans. The experiments in this section are conducted over a given set of concurrent SyncSQL queries. We first enumerate several execution pipelines for the given set of queries. Then, we estimate the cost of executing the different pipelines while changing the following parameters: the input update pattern, the input data distribution, the synchronization period, and the number of queries. Next, we run the query pipelines in the Nile-SyncSQL prototype and measure the execution times. The cost model is verified by matching the measured results with the estimated results.

10.5.1 *Workload Queries and Plan Enumeration.* Experiments in this section are conducted over a set of Group-by queries from the temperature-monitoring application. The goal of these experiments is to illustrate the benefits of using views as a means for the shared execution of continuous queries. The results in this section are conducted from the shared execution of two queries. Including more queries is straightforward. Assume that we have the following two queries:

—`BuildingGroups`: For each building, find the number of rooms with temperature greater than 80. Report modifications in the answer every `i` time units.

—`TemperatureGroups`: For each temperature value `t` that is greater than 80, find the number of rooms that have `t` as the room's temperature. Report modifications in the answer every `j` time units.

Both the `BuildingGroups` and the `TemperatureGroups` queries are aggregate queries over the input stream `RoomTempStr`. However, the `BuildingGroups` query groups the input stream tuples based on the `Building` attribute while the `TemperatureGroups` query groups the input tuples based on the `Temperature` attribute. Also, the two queries differ in the refresh granularity (i.e., require different synchronization streams). Notice that since the two queries are executed over the same input stream (i.e., `RoomTempStr`), it is worth to explore options for sharing the execution of the two queries. We can think of four possible pipelines to execute the two queries concurrently as follows (the corresponding query pipelines are given in Figure 31):

(1) **Non-shared execution:** where the two queries are executed independently without sharing any operations as shown in Figure 31a.

(2) **Shared synchronization:** Figure 31b gives a shared pipeline where a Synchronizer operator is shared between the two queries. The shared Synchronizer uses a synchronization stream that represents the union of the two queries' synchronization streams (i.e., $\text{Sync}_i \cup \text{Sync}_j$).

(3) **Shared pre-aggregation:** Figure 31c gives another shared pipeline where a view is defined and then is used as input to both the `BuildingGroups` and `TemperatureGroups` queries. The shared view consists of a Group-by operator (the operator that is labeled as "GBY:Building, Temperature" in Figure 31c) that groups the input tuples based on both the `Building` and `Temperature` attributes and counts the number of tuples in each group. The output groups from the shared view are then aggregated by the upper Group-by operators

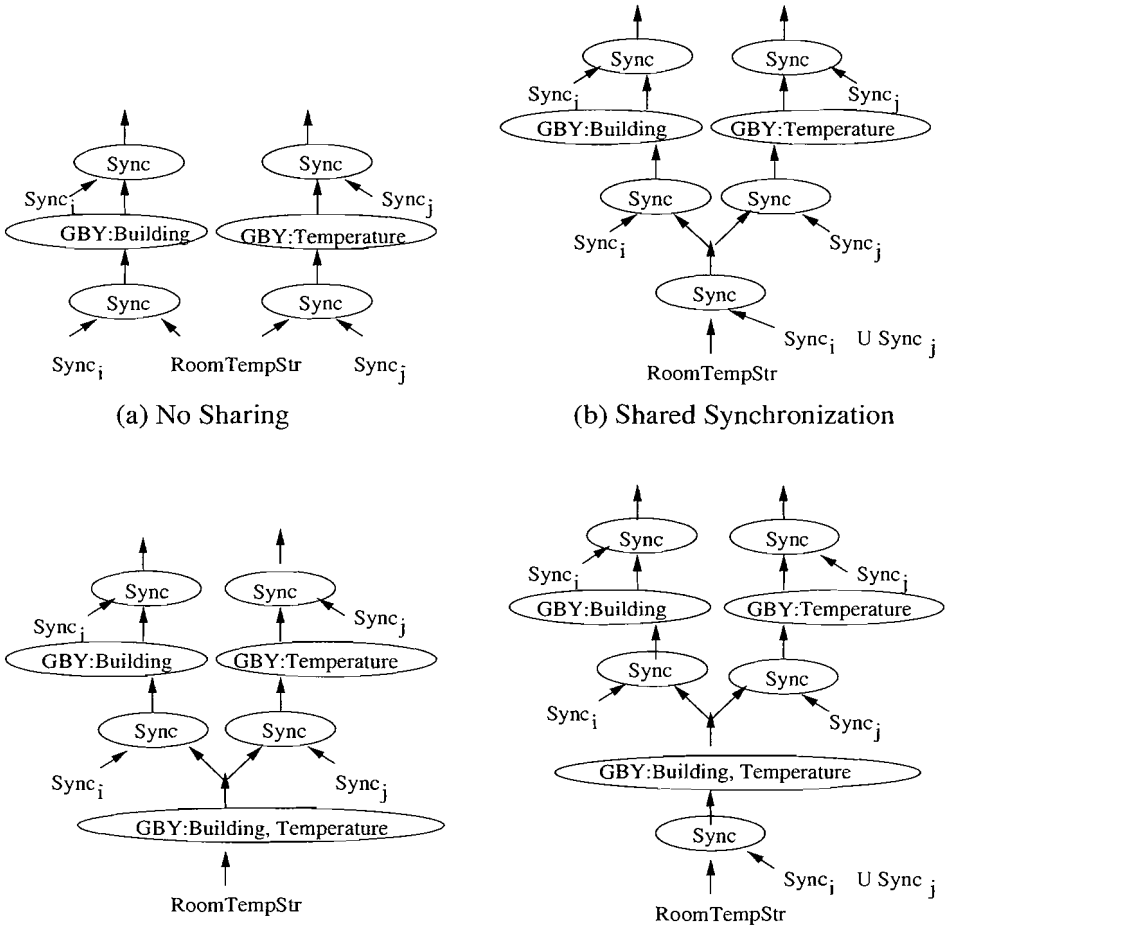


Fig. 31. Possible Execution Plans for Two Concurrent Aggregate Queries.

(the operators that are labeled by "GBY:Building" and "GBY:Temperature" in Figure 31)c to produce the required building and temperature groups. Notice that the "GBY:Building" and "GBY:Temperature" operators sums the number of tuples in the sub-groups to produce the count of tuples in the final group. Notice also that the output stream from the "GBY:Building, Temperature" operator has a primary key that consists of two attributes, namely the Building and Temperature attributes. If the number of `Building-Temperature` groups is less than the number of rooms, then the number of tuples processed by the upper Group-by operators is less than those of the corresponding operators in Pipeline b. However, the shared aggregate operator is considered an additional overhead in Pipeline c.

(4) **Shared synchronization and pre-aggregation:** Figure 31d gives another shared pipeline where both aggregation and synchronization are shared between

the two queries. A shared view is defined that consists of two operators, namely a Synchronizer operator and an aggregate operator. The shared view's Synchronizer operator uses a synchronization stream that represents the union of the two queries's synchronization streams. At the same time, the shared views' aggregate operator groups the input tuples based on both the `Building` and `Temperature` attributes. The benefits of sharing the pre-aggregation is similar to that of case 3. Moreover, as explained in Section 0??, the shared Synchronizer operator performs pre-synchronization and hence reduces the number of input tuple to the view's aggregate operator.

The pipelines in Figure 31 consist of two types of operators, Synchronizer and Group-by operators. Notice that two synchronizer operators are used with each aggregate operator to apply the pre-synchronization optimization as described in Section 0??. In order to estimate the cost of executing a pipeline, we need to estimate two numbers as follows: (1) NS: the number of tuples processed by the Synchronizer operators, and (2) NG: the number of tuples processed by the Group-by operators. Assume that the cost of processing one tuple in any Synchronizer operator equals c_1 while the cost of executing one tuple in any Group-by operator equals c_2. Hence, using the equations in Section 0??, the cost of executing any of the pipelines can be estimated by the following equation:

$$C_{Pipeline} = NS * c_1 + NG * c_2$$

The values of NS and NG differ from one pipeline to another and depend on the following parameters: (1) the update pattern of the input streams, (2) the number of key values in the input streams, (3) the number of Groups that are produced by the Group-by operators, and (4) the synchronization periods. In the following section, we study the effect of the various parameters on the execution cost of the various pipelines.

10.5.2 *Improving the Performance using Views.* In this section, we study the effect of using views as a means for the shared execution of continuous queries. We run an experiment to compare the performance of the non-shared execution pipeline in Figure 31a and the shared execution pipeline in Figure 31d. The parameters for this experiment are as follows:

—Number of rooms is 2000, number of buildings is 20, the number of different temperature values is 10. As a result, the maximum possible number of building-temperature groups is 200.

—Rooms report temperature updates in a uniform pattern where each room reports an update every 1 time unit.

—`BuildingGroups`'s synchronization is every 12 time units while `TemperatureGroups`'s synchronization is every 15 time units.

Using the cost model that is presented in Section 9 and using a similar analysis to that of Figure 16, the execution costs of running Pipelines a and d for 650 time units can be estimated by the following equations:

—$C_a(650) = 2798000 * c_1 + 198000 * c_2$

—$C_d(650) = 1841300 * c_1 + 191800 * c_2$

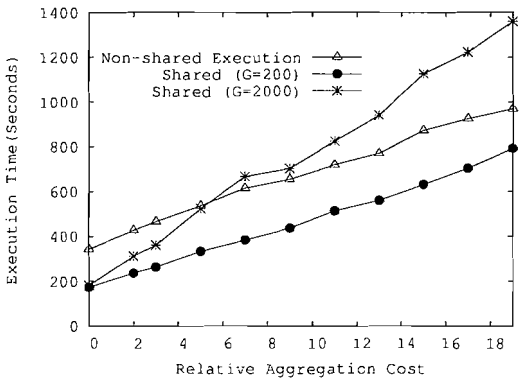


Fig. 32. Non-shared vs. Shared Execution of Aggregate Queries.

The cost equations show that the shared execution in Pipeline d causes 40% reduction in NS and 10% reduction in NG. The reason for the reduction in NS is that, in Pipeline d, the input tuples are processed by only one Synchronizer operator (i.e., the shared Synchronizer operator) in contrast to being processed twice in Pipeline a. The reason for the reduction in NG is that in Pipeline d, the upper Group-by operators process only one tuple for each building-temperature group at every synchronization point in contrast to processing one tuple for each room in Pipeline a. Notice that in this experiment the update rate of the objects (i.e., every 1 time unit) is much higher than the synchronization points rate. This means that at every synchronization point, several updates for the same object are accumulated, hence causing a big reduction in the number of tuples that are processed by the upper operators in the query pipeline.

Effect of the Grouping Factor: If we change the input parameters such that there are 200 buildings, then the number of building-temperature groups can reach up to 2000 at which the cost of Pipeline a is not affected while the cost of Pipeline d is estimated by the following equations:

—$C_d(650) = 2069000 * c_1 + 370000 * c_2$

When the number of building-temperature groups is 2000, the shared execution pipeline consumes 25% less synchronization operations and 1.8% more aggregations than the non-shared execution Pipeline a. Hence the preference between the two pipelines depends on the values of c_1 and c_2.

Figure 32 gives a comparison of the execution times of Pipeline a, Pipeline d with 200 building-temperature groups, and Pipeline d with 2000 building-temperature groups while changing the cost of aggregation (i.e., changing c_2's value). The experimental results show that when the number of groups is 200, the shared execution can achieve up to 50% savings in the execution time than the non-shared execution. However, when the number of groups is 2000, the shared execution performs better than the non-shared execution only for small values of c_2 (i.e., for cheap aggregate functions). As the cost of aggregation increases, the execution time of the shared pipeline increases and the non-shared execution is preferred since it can achieve up to 70% reduction in the execution time. The conclusion from this experiment is

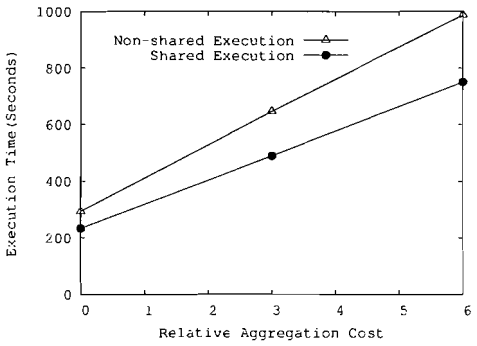


Fig. 33. Effect of the Input Parameters.

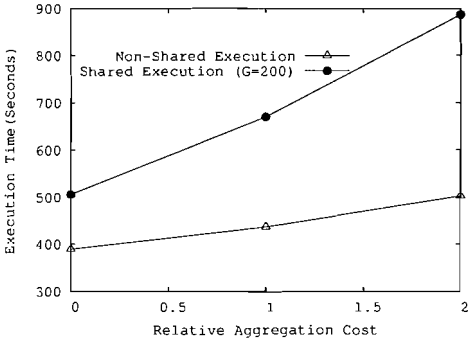


Fig. 34. Effect of using Views on Execution Time.

that the preference between sharing the execution or not depends on (1) the grouping factor (i.e., in the number of groups in each Group-by operator), and (2) the cost of the aggregation function.

Effect of the Input Parameters: The experiment in this section illustrates the effect of the input parameters on the performance. Assume that we run the same experiment as before but with the following parameters:

—Number of rooms is 5000, number of buildings is 100, number of different temperature values is 20. As a result, the maximum number of building-temperature groups is 2000.

—Rooms report temperature updates in a uniform pattern but different rooms have different intervals between the updates as follows. 2500 rooms each reports an update every 2 time units, 1500 rooms each reports an update every 10 time units, and 100 rooms each reports an update every 15 time units.

—`BuildingGroups`'s synchronization is every 6 time units while `TemperatureGroups`'s synchronization is every 12 time units.

Figure 33 illustrates that the shared execution pipeline improves the execution time over that of the independent execution pipeline. However, the percentage of execution time reduction is less than that in Figure 32. The percentage of reduction in execution time is 40% in contrast to 70% in Figure 32. The reason for the difference in the performance gain is that in the earlier parameters settings the update rate is much higher than the frequency of the synchronization points. However, in the parameters in this section, the synchronization points are as frequent as the object update rate. Hence, not too many updates are accumulated by the Synchronizer operator. As a result, synchronization has only a small effect on the number of tuples that flow in the query pipeline.

10.5.3 *Can Views Worsen the Performance?.* The experiments of the previous section illustrate that using views improves the query performance. However, the improvement factor depends on the query settings. In this section, we show that for some input parameters, using views may worsen the query performance. Consider

the two queries BuildingGroups and TemperatureGroups with the following input parameters:

—Number of rooms is 2000, number of buildings is 100, number of different temperature values is 20. As a result, the number of building-temperature groups is 2000.

—Rooms report temperature updates in a uniform pattern but with different intervals as follows. 500 rooms each reports an update every 10 time units, 500 rooms each reports an update every 13 time units, and 1000 rooms each reports an update every 17 time units.

—BuildingGroups's synchronization is every 15 time units while TemperatureGroups's synchronization is every 12 time units.

Using the proposed cost model and the equation that is presented in Section 10.5.1, the execution cost of the non-shared execution pipeline (Pipeline a) and the shared execution pipeline (Pipeline d) for 650 time units can be estimated by the following equations:

—$C_a(650) = 444924 * c_1 + 171886 * c_2$

—$C_d(650) = 743007 * c_1 + 288318 * c_2$

The cost equations show that the shared execution paradigm requires more synchronization operations and more aggregation operations than the non-shared execution. As a result, in this case, non-shared execution is always preferred over shared execution. The analytical results are confirmed by the experimental results that are given in Figure 34. The graphs in Figure 34 illustrate that non-shared execution achieves up to 50% reduction in the execution time. The reason for the winning performance of non-shared execution is that the update rates for most of the rooms are slower than that of the synchronization rate. As a result, synchronization does not result in any accumulation of updates and hence does not reduce the number of tuples to be processed by the query pipeline. Moreover, the number of intermediate building-temperature groups is the same as the number of rooms. Hence, shared execution does not result in any reduction in the number of tuples. The shared view is nothing but an additional overhead, hence causes bad execution times.

10.5.4 *Effect of the Number of Queries.* In this section we run experiments to study the effect of the number of queries that share the view execution on the performance. Assume that we have several queries that are similar to the BuildingGroups query in that they group on the building attribute. However, the queries differ in the synchronization streams. Similarly, assume that there are several queries that are similar to the TemperatureGroups query but with different synchronization streams. Figure 35 gives the effect of the number of queries on the execution time. The results in Figure 35 illustrate that sharing the execution of the view between two queries improves the execution time by around 25%. However, sharing the execution of the view among 5 queries improves the execution time by up to 70%. The conclusion from this experiment is that the more the queries that utilize the view, the more the improvement in the performance.

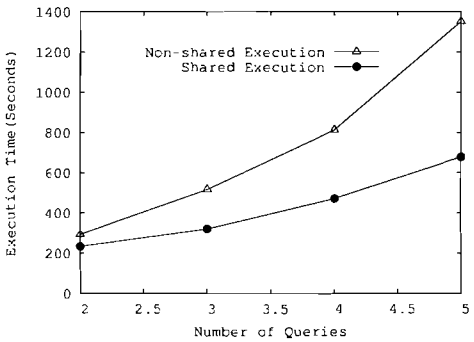


Fig. 35. Effect of Number of Queries on Execution Time.

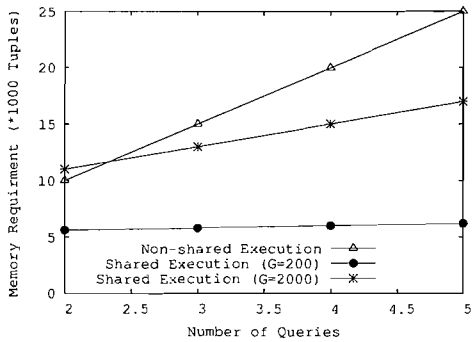


Fig. 36. Effect of Number of Queries on Memory.

10.5.5 *Memory Requirements.* As discussed in Section 0??, some query operators need to maintain a state in order to continuously keep track of the query answer. The summation of the state sizes of the various operators represents the memory requirements for a given execution pipeline. For example, for the execution pipeline that is given in Figure 31d, the memory requirement is calculated as the summation of the sizes of the states that are maintained by the three Group-by operators (namely the GBY:Building, GBY:Temperature, and GBY:Building,Temperature operators) and the five Synchronizer operators. Notice that the Synchronizer's state size is proportional to the number of distinct key values in the input stream. However, the size of the Group-by's state is proportional to the number of groups.

Figure 36 gives the effect of the number of queries on the memory requirement. Assume that initially we have two queries, namely the `BuildingGroups` and the `TemperatureGroups` queries. Assume further that we add queries that are similar to either `BuildingGroups` or `TemperatureGroups` but with different synchronization stream. The number of rooms in this experiment is set to 5000 rooms. Notice that one Group-by and two Synchronizer operators are added for each additional query. However, the states that are needed by the additional operators depend on the execution paradigm. Figure 36 gives a comparison of the memory requirements for the following three execution pipelines: non-shared execution, shared execution with 200 building-temperature groups, and shared execution with 2000 building-temperature groups. The graphs in Figure 36 illustrate that the non-shared execution requires more memory and the size of the additional memory is proportional to the number of queries. The reason is that an additional Synchronizer operator over the input stream, with a state of size 5000, is added for each additional query. However, in the case of shared execution, an additional synchronizer operator is added with a state size equals to the number of building-temperature groups (i.e., 200 or 2000).

11. CONCLUSIONS

REFERENCES

http://www.streambase.com.

ABADI, D., AHMAD, Y., BALAKRISHNAN, H., BALAZINSKA, M., CETINTEMEL, U., CHERNIACK, M., HWANG, J.-H., JANOTTI, J., LINDNER, W., MADDEN, S., RASIN, A., STONEBRAKER, M., TATBUL, N., XING, Y., AND ZDONIK, S. 2005. The design of the borealis stream processing engine. In *Proceedings of the International Conference of Innovative Data Systems Research, CIDR.*

ABADI, D. J., CARNEY, D., CETINTEMEL, U., CHERNIACK, M., CONVEY, C., LEE, S., STONEBRAKER, M., TATBUL, N., AND ZDONIK, S. B. 2003. Aurora: A new model and architecture for data stream management. *The International Journal on Very Large Data Bases, VLDB Journal 12,* 2, 120–139.

AHRENS, J. AND DIETER, U. 1974. Computer methods for sampling from gamma, beta, poisson, and binomial distributions. *Computing 12,* 3, 223–246.

ARASU, A., BABU, S., AND WIDOM, J. 2006. The CQL continuous query language: Semantic foundations and query execution. *The International Journal on Very Large Data Bases, VLDB Journal 15,* 2, 121–142.

ARASU, A. AND WIDOM, J. 2004a. A denotational semantics for continuous queries over streams and relations. *SIGMOD Record 33,* 3, 6–12.

ARASU, A. AND WIDOM, J. 2004b. Resource sharing in continuous sliding-window qggregates. In *Proceedings of the International Conference on Very Large Data Bases, VLDB.*

BABCOCK, B., BABU, S., DATAR, M., MOTWANI, R., AND WIDOM, J. 2002. Models and issues in data streams. In *Proceedings of the ACM Symposium on Principles of Database Systems, PODS.*

BABU, S., MUNAGALA, K., WIDOM, J., AND MOTWANI, R. 2005. Adaptive caching for continuous queries. In *Proceedings of the International Conference on Data Engineering, ICDE.*

BONNET, P., GEHRKE, J. E., AND SESHADRI, P. 2001. Towards sensor database systems. In *Proceesings of the Mobile Data Management Conference, MDM.*

CAREY, M. J., DEWITT, D. J., FRANKLIN, M. J., HALL, N. E., MCAULIFFE, M. L., NAUGHTON, J. F., SCHUH, D. T., SOLOMON, M. H., TAN, C. K., TSATALOS, O. G., WHITE, S. J., AND ZWILLING, M. J. 1994. Shoring up presistent applications. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD.*

CARNEY, D., CETINTEMEL, U., CHERNIACK, M., CONVEY, C., LEE, S., SEIDMAN, G., STONEBRAKER, M., TATBUL, N., AND ZDONIK, S. B. 2002. Monitoring streams - A new class of data management applications. In *Proceedings of the International Conference on Very Large Data Bases, VLDB.*

CHANDRASEKARAN, S., COOPER, O., DESHPANDE, A., FRANKLIN, M. J., HELLERSTEIN, J. M., HONG, W., KRISHNAMURTHY, S., MADDEN, S., RAMAN, V., REISS, F., AND SHAH, M. A. 2003. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Proceedings of the International Conference of Innovative Data Systems Research, CIDR.*

CHANDRASEKARAN, S. AND FRANKLIN, M. J. 2003. PSoup: A system for streaming queries over streaming data. *The International Journal on Very Large Data Bases, VLDB Journal 12,* 2, 140–156.

CHEN, J., DEWITT, D. J., TIAN, F., AND WANG, Y. 2000. NiagaraCQ: A scalable continuous query system for internet databases. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD.*

CRANOR, C. D., JOHNSON, T., SPATSCHECK, O., AND SHKAPENYUK, V. 2003. Gigascope: A stream database for network applications. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD.*

EISENBERG, A., MELTON, J., KULKARNI, K., MICHELS, J.-E., AND ZEMKE, F. 2004. SQL:2003 has been published. *SIGMOD Record 33,* 1, 119–126.

ESL. An introduction to the expressive stream language. WEB Information System Laboratory, UCLA, CS Department. http://wis.cs.ucla.edu/stream-mill.

GAMA, J. AND GABER, M. M., Eds. 2007. *Data stream processing techniques in sensor networks.* Springer, Chapter Data stream management systems and architectures.

GANGULY, S., GAROFALAKIS, M., AND R.RASTOGI. 2003. Processing set expressions over continuous update streams. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD.*

GHANEM, T. M., AREF, W. G., AND ELMAGARMID, A. K. 2006. Exploiting predicate-window semantics over data streams. *SIGMOD Record 35,* 1, 3–8.

GHANEM, T. M., HAMMAD, M. A., F.MOKBEL, M., AREF, W. G., AND ELMAGARMID, A. K. 2007. Incremental evaluation of sliding-window queries over data streams. *IEEE Transactions on Knowledge and Data Engineering, TKDE 19,* 1, 57–72.

GOLAB, L. AND OZSU, M. T. 2003. Issues in data stream management. *SIGMOD Record 32,* 2, 5–14.

GOLDSTEIN, J. AND LARSON, P.-Å. 2001. Optimizing queries using materialized views: A practical, scalable solution. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD.*

GRIFFIN, T. AND LIBKIN, L. 1995. Incremental maintenance of views with duplicates. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD.*

GUPTA, A. AND MUMICK, I. S., Eds. 1999. *Materialized views: Techniques, implementation, and applications.* MIT Press.

HALEVY, A. Y. 2001. Answering queries using views: A survey. *The International Journal on Very Large Data Bases, VLDB Journal 10,* 4, 270–294.

HAMMAD, M. A., FRANKLIN, M. J., AREF, W. G., , AND ELMAGARMID, A. K. 2003. Scheduling for shared window joins over data streams. In *Proceedings of the International Conference on Very Large Data Bases, VLDB.*

HAMMAD, M. A., M.GHANEM, T., AREF, W. G., ELMAGARMID, A. K., AND F.MOKBEL, M. 2004. Efficient pipelined execution of sliding window queries over data streams. In *Purdue University Technical Report, CSD TR 03-035.*

HAMMAD, M. A., MOKBEL, M. F., ALI, M. H., AREF, W. G., CATLIN, A. C., ELMAGARMID, A. K., ELTABAKH, M., ELFEKY, M. G., GHANEM, T. M., GWADERA, R., ILYAS, I. F., MARZOUK, M., AND XIONG, X. 2004. Nile: A query processing engine for data streams (demo). In *Proceedings of the International Conference on Data Engineering, ICDE.*

KANG, J., NAUGHTON, J. F., AND VIGLAS, S. 2003. Evaluating window joins over unbounded streams. In *Proceedings of the International Conference on Data Engineering, ICDE.*

LARSON, P.-Å. AND YANG, H. Z. 1985. Computing queries from derived relations. In *Proceedings of the International Conference on Very Large Data Bases, VLDB.*

LERNER, A. AND SHASHA, D. 2003. The virtues and challenges of ad hoc + streams querying in finance. *IEEE Data Engineering Bulletin 26,* 1, 49–56.

LI, J., MAIER, D., TUFTE, K., PAPADIMOS, V., AND TUCKER, P. 2005. Semantics and evaluation techniques for window aggregates in data streams . In *Proceedings of the ACM International Conference on Management of Data, SIGMOD.*

LIU, L., PU, C., AND TANG, W. 1999. Continual queries for internet scale event-driven information delivery. *IEEE Transactions on Knowledge and Data Engineering, TKDE 11,* 4, 610–628.

MAIER, D., LI, J., TUCKER, P., TUFTE, K., AND PAPADIMOS, V. 2005. Semantics of data streams and operators. In *Proceedings of the International Conference on Database Theory, ICDT.*

MOTWANI, R., WIDOM, J., ARASU, A., BABCOCK, B., BABU, S., DATAR, M., MANKU, G. S., OLSTON, C., ROSENSTEIN, J., AND VARMA, R. 2003. Query processing, approximation, and resource management in a data stream management system. In *Proceedings of the International Conference of Innovative Data Systems Research, CIDR.*

PERVASIVE INFRASTRUCTURE SENSOR NETWORKS. http://www.sensornets.org/.

RYVKINA, E., MASKEY, A. S., CHERNIACK, M., AND ZDONIK2, S. 2006. Revision processing in a stream processing engine: A high-level design. In *Proceedings of the International Conference on Data Engineering, ICDE.*

SESHADRI, P. AND PASKIN, M. 1997. PREDATOR: An OR-DBMS with enhanced data types. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD.*

SNODGRASS, R. AND AHN, I. 1985. A taxonomy of time in databases. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD.*

SRIVASTAVA, U. AND WIDOM., J. 2004. Flexible time management in data stream systems. In *Proceedings of the ACM Symposium on Principles of Database Systems, PODS.*

STONEBRAKER, M., CETINTEMEL, U., AND ZDONIK, S. 2005. The 8 requirements of real-time stream processing. *SIGMOD Record 34*, 4, 42–47.

STREAMSQL. http://www.StreamSQL.org.

SULLIVAN, M. AND HEYBEY, A. 1998. Tribeca: A System for managing large databases of netwrok traffic. In *USENIX*.

SZEWCZYK, R., OSTERWEIL, E., POLASTRE, J., HAMILTON, M., MAINWARING, A. M., AND ESTRIN, D. 2004. Habitat monitoring with sensor networks. *Communications of the ACM 47*, 6, 34–40.

TERRY, D. B., GOLDBERG, D., NICHOLS, D., AND OKI, B. M. 1992. Continuous queries over append-only databases. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*.

YAN, W. P. AND LARSON, P.-A. 1995. Eager aggregation and lazy aggregation. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*.

YAO, Y. AND GEHRKE, J. 2003. Query processing for sensor networks. In *Proceedings of the International Conference of Innovative Data Systems Research, CIDR*.